目錄

操作系统思考 中文版

作者：Allen B. Downey

原文：Think OS: A Brief Introduction to Operating Systems

译者：飞龙

版本：0.5.0

- 在线阅读
- PDF格式
- EPUB格式
- MOBI格式
- Github

赞助我

协议

第零章 前言

> 作者：Allen B. Downey
>
> 原文：Chapter 0 Preface
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0

在许多计算机科学的课程中，操作系统都是高级话题。学生在上这门课之前，它们已经知道了如何使用C语言编程，他们也可能上过计算机体系结构（组成原理）的课程。通常这门课的目标是让学生们接触操作系统的设计与实现，并带有一些他们未来在该领域所研究的隐含假设，或者让他们手写OS的一部分。

这本书为一些不同的读者准备，并且具有不同的目标。我为欧林工学院中一门叫做软件系统的课程编写了它。

多数学生在学完Python编程之后上了这门课，所以目标之一就是帮助他们学习C语言。对于课程的这一部分，我使用了O’Reilly的《Head First C》（中译本为《嗨翻C语言》）作为补充。

我的一些学生从没有写过操作系统，但是它们中许多人都会使用C语言编写底层的应用，或者与嵌入式打交道。我的课程包括操作系统的要素、网络、数据库、和嵌入式系统，而且强调了程序员需要知道的一些话题。

这本书并不假设你学过计算机体系结构。在讲解过程中，我会解释所需的东西。

如果这本书成功了，它会带给你对程序运行中所发生事情的深入理解，并且你可以使它们运行速度更快以及更加可靠。

第一章解释了编译语言和解释语言的一些差异，以及编译器工作原理的一些洞察。推荐阅读《嗨翻C语言》的第一章。

第二章解释了操作系统如何使用进程来保证运行中的程序不相互影响。

第三章解释了虚拟内存和地址翻译。推荐阅读《嗨翻C语言》的第二章。

第四章有关文件系统和数据流。推荐阅读《嗨翻C语言》的第三章。

第五章描述了数值、字母和其它值如何编码，同时展示了按位运算。

第六章解释了如何使用动态内存管理，它如何工作。推荐阅读《嗨翻C语言》的第六章。

第七章有关缓存和存储器层次结构。

第八章有关多任务和调度。

第九章有关POSIX线程和互斥体。推荐阅读《嗨翻C语言》的第十二章，和《Little Book of Semaphores》的第一和第二章。

第十章有关POSIX条件变量和生产者/消费者问题。推荐阅读《Little Book of Semaphores》的第三和第四章。

第十一章有关POSIX信号量和C中的实现。

这份草稿的注解

本书的当前版本（v0.5）是个初稿。当我处理文字时，我还没有把图片放进来。所以我确信有些地方的解释加上图片之后会更好。

0.1 代码的使用

本书的示例代码可以在https://github.com/AllenDowney/ThinkOS访问。Git是一个版本控制系统，它允许你跟踪项目所组成的文件。Git控制下的一系列文件叫做仓库。GitHub是一个为Git仓库提供储存空间的托管服务，以及一个便利的Web界面。

我的仓库的GitHub的主页提供了如下方式来获取代码：

- 你可以通过点击"Fork"按钮，在GitHub上创建我的仓库的一份副本。如果你没有GitHub账号，你需要创建一个。在Fork之后，你在GitHub上就有了自己的仓库，你可以在本书编写的过程中，将其用于跟踪你编写的代码。之后你可以克隆这个仓库，也就是说你可以将文件复制到自己的电脑上。
- 或者你可以克隆我的仓库。你并不需要GitHub账号来完成它，但是你不能将你的修改写回GitHub。
- 如果你完全不想使用Git，你可以使用GitHub页面右下角的按钮，下载以Zip打包的文件。

贡献者名单

如果你需要提供建议或纠错，请向downey@allendowney.com发送邮件。如果我基于你的反馈作出修改，我会将你添加到贡献者名单中（除非你要求被忽略）。

如果你包含了错误所在句子的一小部分，我会很容易找到它。页面和章节的号码也可以，但是不是十分易于处理。多谢了！

- 我要感谢欧林工学院软件系统课上的所有学生，他们在2014春季学期测试了这本书的初稿。他们纠正了许多错误，并提了很多有用的建议。我很欣赏他们的开拓精神！
- Donald Robertson 指出了两个打字错误。
- Jim Tyson 提交了两个纠正。
- James P Giannoules 指出了一处复制粘贴错误。

- Andy Engle 给出了GB和GiB的差异。
- Aashish Karki 指出了一些错误的语法。

第一章 编译

> 作者：Allen B. Downey
>
> 原文：Chapter 1 Compilation
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0

1.1 编译语言和解释语言

人们通常把编程语言描述为编译语言或者解释语言。前者的意思是程序被翻译成机器语言，之后由硬件执行；而后者的意思是程序被软件解释器读取并执行。例如，C被认为是编译语言，而Python被认为是解释语言。但是二者之间的界限并不总是那么明显。

首先，许多语言既可以编译执行也可以解释执行。例如，存在C的解释器，和Python的编译器。其次，类似Java的语言混合了这两种方法，它先把程序编译成中间语言，之后在解释器中执行转换后的程序。Java使用了一种叫做“Java 字节码”的中间语言，它类似于机器语言，但是由软件解释器执行，即Java虚拟机（JVM）。

所以，编译执行或解释执行并不是语言的内在特征。尽管如此，在编译语言和解释语言之间有一些普遍的差异。

1.2 静态类型

许多解释语言都支持动态类型，但是编译语言通常限制为静态类型。在静态类型的语言中，你可以通过观察程序，来分辨出每个变量都指向哪种类型。在动态类型的语言中，直到运行起来，你才能知道变量的类型。通常，“静态”指那些在编译时发生的事情，而“动态”指在运行时发生的事情。

例如，在Python中你可以像这样编写函数：

```
def add(x, y):
    return x + y
```

观察这段代码，你不能分辨出 `x` 和 `y` 所指向的类型。这个函数在运行时可能会调用数次，每次都接受不同类型的值。任何支持加法操作的值都是有效的，任何其它类型的值都会引发异常，或者“运行时错误”。

C中你可以像这样编写同样的函数：

```
int add(int x, int y) {
    return x + y;
}
```

函数的第一行包含了参数及返回值的“类型声明”： `x` 和 `y` 都声明为整数，这意味着我们可以在编译时检查加法操作对该类型是否合法（是的）。返回值也声明为整数。

由于这些类型声明，当函数在程序其它位置调用时，编译器就可以检查所提供的参数是否具有正确类型，以及返回值是否使用正确。

这些检查在程序开始运行之前发生，所以可以更快地找到错误。更重要的是，程序永远不会运行的一部分中也可以找到错误。而且，这些检查不必发生于运行期间，这也是编译语言通常快于解释语言的原因之一。

编译时的类型声明也会节省空间。在动态语言中，变量的名称在程序运行时储存在内存中，并且它们通常可由程序访问。例如，在Python中，内建的 `locals` 函数返回含有变量名称和值的字典。下面是Python解释器中的一个示例：

```
>>> x = 5
>>> print locals()
{'x': 5, '__builtins__': <module '__builtin__' (built-in)>,
'__name__': '__main__', '__doc__': None, '__package__': None}
```

这段代码表明，变量的名称在程序运行期间储存在内存中（以及其它作为默认运行时环境一部分的值）。

在编译语言中，变量的名称只存在于编译时，而不是运行时。编译器为每个变量选择一个位置，并记录这些位置作为所编译程序的一部分[1]。变量的位置被称为“地址”。在运行期间，每个变量的值都储存在它的地址处，但是变量的名称完全不会储存（除非它们由于调试目的被编译器添加）。

> [1] 这只是一个简述，之后我们会深入了解更多细节。

1.3 编译过程

作为程序员，你应该对编译期间发生的事情有所认识。如果你理解了这个过程，它会帮助你解释错误信息，调试你的代码，以及避免常见的陷阱。

下面是编译的步骤：

1. 预处理：C是包含“预处理指令”的几种语言之一，它生效于编译之前。例如， `#include` 指令使其它文件的源代码插入到指令所在的位置。
2. 解析：在解析过程中，编译器读取源代码，并构建程序的内部表示，称为“抽象语法树”（AST）。这一阶段的错误检测通常为语法错误。
3. 静态检查：编译器会检查变量和值的类型是否正确，函数调用是否带有正确数量和类型

的参数，以及其它。这一阶段的错误检测通常为一些“静态语义”的错误。
4. 代码生成：编译器读取程序的内部表示，并生成机器码或字节码。
5. 链接：如果程序使用了定义在库中的值或函数，编译器需要找到合适的库并包含所需的代码。
6. 优化：在这个过程的几个时间点上，编译器可以修改程序来生成运行更快或占用更少空间的代码。大多数优化都是一些简单的修改，来消除明显的浪费。但是一些编译器会执行复杂的分析和修改。

通常当你运行 `gcc` 时，它会执行上述所有步骤，并且生成一份可执行文件。例如，下面是一个小型的C语言程序：

```
#include <stdio.h>
int main()
{
    printf("Hello World\n");
}
```

如果你把它保存在名为 `hello.c` 的文件中，你可以像这样编译并运行它：

```
$ gcc hello.c
$ ./a.out
```

通常， `gcc` 将可执行代码储存在名为 `a.out` 的文件中（它原本代表汇编器的输出，即“assembler output”）。第二行运行了这个可执行文件。 `./` 前缀告诉shell在当前目录中寻找它。

使用 `-o` 选项来为可执行文件提供一个更好的名字，通常是个不错的注意。

```
$ gcc hello.c -o hello
$ ./hello
```

1.4 目标代码

`-c` 选项告诉 `gcc` 编译程序并生成机器码，但是不链接它们或生成可执行文件：

```
$ gcc hello.c -c
```

执行结果是名为 `hello.o` 的文件，其中 `o` 代表“目标代码”（object code），它就是编译后的程序。目标代码并不是可执行代码，但是它可以链接到可执行文件中。

`nm` UNIX命令可以读取目标文件并生成关于它所定义和所使用的名称的信息。例如：

```
$ nm hello.o
0000000000000000 T main
                 U puts
```

输出显示，`hello.o` 定义了 `main` 名称，并使用了 `puts` 函数，它代表“输出字符串”（put string）。在这个例子中，`gcc` 通过将 `printf` 替换掉执行了优化，它是一个复杂的大型函数。而 `puts` 相对来说比较简单。

你可以使用 `-O` 选项来控制 `gcc` 优化的程度。通常，它执行非常细微的优化，可以使调试更加容易。 `-O1` 选项会开启最为普通和安全的优化。更高的数值开启需要长时间编译的高级优化。

理论上，优化除了加速运行之外，不应改变程序的行为。但是如果你的程序中有微妙的bug，你可能会发现，优化会使bug出现或消失。在开发新的代码时，关闭优化通常是一个不错的注意。一旦程序正常运行并通过了适当的测试，你可以开启优化，并确保测试仍然能够通过。

1.5 汇编代码

和 `-c` 选项类似。 `-S` 告诉 `gcc` 编译程序并生成汇编代码，它通常为机器代码的可读形式。

```
$ gcc hello.c -S
```

执行结果是名为 `hello.s` 的文件，它可能看起来是这样：

```
        .file        "hello.c"
        .section     .rodata
.LC0:
        .string      "Hello World"
        .text
        .globl       main
        .type        main, @function
main:
.LFB0:
        .cfi_startproc
        pushq %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq %rsp, %rbp
        .cfi_def_cfa_register 6
        movl $.LC0, %edi
        call puts
        movl $0, %eax
        popq %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size        main, .-main
        .ident       "GCC: (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3"
        .section     .note.GNU-stack,"",@progbits
```

`gcc` 通常为你所运行的机器生成代码，所以对我来说它生成x86汇编代码，运行在Intel、AMD和许多其它处理器上面。如果你运行在不同的架构上，你会看到不同的代码。

1.6 预处理

在编译过程中再往前退一步，你可以使用`-E`选项来只运行预处理器：

```
$ gcc hello.c -E
```

执行结果就是预处理器的输出。这个例子中，它含有来自`stdio.h`的被包含代码，和`stdio.h`所包含的所有文件，还有这些文件所包含的所有文件，以及其它。在我的机器上，共计800行代码。因为几乎每个C语言程序都会包含`stdio.h`，这800行代码经常会被编译。如果你像大多数C程序那样也包含了`stdlib.h`，结果会变成多于1800行代码。

1.7 理解错误

既然我们知道了编译过程的步骤，理解错误消息就变得十分容易。例如，如果`#include`指令中出现了一个错误，你会从预处理器处得到一个错误：

```
hello.c:1:20: fatal error: stdioo.h: No such file or directory
compilation terminated.
```

如果有语法错误，你会从编译器处得到一个错误：

```
hello.c: In function 'main':
hello.c:6:1: error: expected ';' before '}' token
```

如果你使用了没有在任何标准库中定义的函数，你会从链接器处得到一个错误：

```
/tmp/cc7iAUbN.o: In function `main':
hello.c:(.text+0xf): undefined reference to `printff'
collect2: error: ld returned 1 exit status
```

`ld`是UNIX链接器的名称，这样命名是因为“装载”（loading）是编译过程中的另一个步骤，它和链接关系密切。

一旦程序运行起来，C会执行非常少的运行时检测，所以你会看到极少的运行时错误。如果你发生了除零错误，或者执行了其它非法的浮点操作，你会得到“浮点数异常”。而且，如果你尝试读写内存的不正确位置，你会得到“段错误”。

第二章 进程

> 作者：Allen B. Downey
>
> 原文：Chapter 2 Processes
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0

2.1 抽象和虚拟化

在我们谈论进程之前，我打算先定义几个东西：

- 抽象（Abstraction）：抽象是复杂事物的简单表示。例如，如果你开车的话，应该知道车轮向左转的时候车也会向左行驶，反之亦然。当然，方向盘由一系列机械和传动系统所连接，用于使轮子转向，并且轮子和路面的相互作用方式也很复杂。但是作为一个司机，你通常不需要考虑这些细节。你可以仅仅建立方向盘的心智模型，这种心智模型就是一个抽象。

  软件工程的很大一部分就是设计类似这样的抽象，允许用户和其它程序员使用强大而复杂的系统，而不必知道其实现的细节。

- 虚拟化（Virtualization）：一类非常重要的抽象就是虚拟化，它是创建可取的幻像的过程。例如，许多公共图书馆都参与了馆际合作，允许它们互相借阅图书。当我需要一本书时，有时它在我的本地图书馆的架子上，但更多情况下它会被运到其它的馆藏中。无论是哪一种，我都会收到它可借阅的提醒。我并不需要知道它来自哪里，我也不需要知道我的图书馆拥有哪一本书。一般来说，这个系统创建了一个幻象，好像我的图书馆拥有全世界的每一本书。

  在物理上，我的图书馆的馆藏可能很小，但是虚拟上我能获得的馆藏包含了馆际合作的每一本书。

  另外一个例子，大多数电脑都只连接到一个网络中，而这个网络又链接到其它网络，等等。我们所谈论的“互联网”，是一系列网络和协议的合集，它将数据包从一个网络传送到另一个网络。从用户和程序员的角度来看，整个系统的行为就像是互联网的每台计算机都互相连接。物理连接的数量十分少，但是虚拟连接的数量十分庞大。

“虚拟”这个词通常用于虚拟机的语境中，它是一种软件，可以创建运行特定系统的专用计算机的幻象。实际上，虚拟机可能和其它虚拟机一起运行在不同的操作系统上。

在虚拟化的语境中，我们通常把真实发生的事情叫做“物理的”，而把虚拟上发生的事情叫做“逻辑的”或者“抽象的”。

2.2 隔离

工程最重要的原则之一就是隔离（Isolation）：当你设计一个带有多个组件的系统时，将它彼此隔离是个很好的方法，这样某个组件中的改变就不会对其它组件造成不良影响。

操作系统最重要的目标之一，就是将每个进程和其它进程隔离，使程序员不必考虑每个可能的交互情况。提供这种隔离的软件对象叫做进程（Process）。

进程是表示运行中程序的软件对象。我按照面向对象编程把它称之为“软件对象”。通常一个对象包含数据，并且提供用于操作数据的方法。进程正是包含以下数据的对象：

- 程序文本，通常是机器语言的指令序列。
- 程序相关的数据，包括静态数据（编译时分配）和动态数据，后者包括运行时的栈和堆。
- 任何等待中的IO状态。例如，如果进程正在等待从磁盘中读取的数据，或者从网络到达的数据包，这些操作的状态也是进程的一部分。
- 程序的硬件状态，这包括储存在寄存器中的数据，状态信息，以及程序计数器，它表示当前执行了哪个指令。

通常一个进程运行一个程序，但是对于进程来说，加载并运行新的程序也是可能的。

也可以在多于一个进程中运行相同的程序，这非常常见。这种情况下，各个进程共享程序文本，但是拥有不同的数据和硬件状态。

大多数操作系统提供了隔离进程的基本功能：

- 多任务：大多数操作系统有能力在几乎任何时候中断一个进程，保存它的硬件状态，并且在以后恢复它。通常，程序员不需要考虑这些中断。程序的行为就像在一个专用的处理器上持续运行，除了两条指令之间的时间是不可预测的。
- 虚拟内存：大多数操作系统会创建幻象，每个进程看似拥有独立内存片并且孤立于其他进程。同样，程序员通常也不需要考虑虚拟内存如何工作，他们可以当做每个程序都拥有专用的内存片来处理。
- 设备抽象：运行于同一台计算机的进程共享磁盘、网络接口、显卡和其它硬件。如果进程直接和这些硬件交互而不加协调，就一定会产生混乱。例如，一个进程预期的网络数据可能会被另一个进程读取。或者多个进程可能尝试在磁盘的相同位置储存数据。操作系统负责通过提供合适的抽象来维持秩序。

作为程序员，你不需要知道太多关于这些功能如何实现的事情。但是如果你很好奇，你可以在这个屏蔽层的后面发现一大堆有趣的事情。而且，如果你知道其中所发生的事情，你会成为更好的程序员。

2.3 Unix 进程

当我写这本书的时候，我最关注的进程就是我的文本编辑器，Emacs。偶尔我也会切换到终端窗口，它是一个运行Unix shell并提供命令行接口的窗口。

当我移动鼠标时，窗口的管理器会被唤醒，看到鼠标在终端窗口上方，并且唤醒终端。终端又唤醒shell。如果我在shell中键入 `make` ，它就会创建一个新的进程来运行Make。Make会创建另一个进程来运行LaTeX，之后另一个进程会显示结果。

如果我需要查询一些东西，我会切换到另一个桌面，这会再次唤醒窗口管理器。如果我点击Web浏览器的图标，窗口管理器会创建进程来运行Web浏览器。许多浏览器，类似Chrome，会为每个窗口和每个选项卡创建新的进程。

并且这些只是我所了解的进程，同时还有许多其它进程“在后台”运行。它们中许多都在执行操作系统相关的工作。

Unix命令 `ps` 能打印出运行中进程的信息。如果你在终端里运行它，可能会看到这些：

```
  PID TTY          TIME CMD
 2687 pts/1    00:00:00 bash
 2801 pts/1    00:01:24 emacs
24762 pts/1    00:00:00 ps
```

第一列是唯一的进程ID。第二列是创建进程的终端，“TTY”代表“电传打字机”（Teletypewriter），它是原始的机械终端。

第三行是用于该进程的处理器时间总计，依次为时、分、秒。最后一行是所运行进程的名称。这个例子中， `bash` 是shell的名称，用于解释我键入到终端中的命令。Emacs是我的文本编辑器，而 `ps` 是生成这份输出的程序。

通常， `ps` 只会列出有关当前终端的进程。如果你使用 `-e` 选项，你会得到所有进程（也包括属于其他用户的进程，我认为这是个安全缺陷）。

在我的系统上有233个进程，下面是它们的一部分：

```
  PID TTY          TIME CMD
    1 ?        00:00:17 init
    2 ?        00:00:00 kthreadd
    3 ?        00:00:02 ksoftirqd/0
    4 ?        00:00:00 kworker/0:0
    8 ?        00:00:00 migration/0
    9 ?        00:00:00 rcu_bh
   10 ?        00:00:16 rcu_sched
   47 ?        00:00:00 cpuset
   48 ?        00:00:00 khelper
   49 ?        00:00:00 kdevtmpfs
   50 ?        00:00:00 netns
   51 ?        00:00:00 bdi-default
   52 ?        00:00:00 kintegrityd
   53 ?        00:00:00 kblockd
   54 ?        00:00:00 ata_sff
   55 ?        00:00:00 khubd
   56 ?        00:00:00 md
   57 ?        00:00:00 devfreq_wq
```

`init` 是操作系统启动时首先创建的进程。它又会创建许多其它进程，之后会闲置，直到它创建的进程运行完毕。

`kthreadd` 是操作系统用于创建新的“线程”的进程。之后我们将会谈论更多关于线程的东西，但是你暂时你可以认为线程是一种进程。

第三章 虚拟内存

> 作者：Allen B. Downey
>
> 原文：Chapter 3 Virtual memory
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0

3.1 简明信息理论

比特是二进制的数字，也是信息的单位。一个比特有两种可能的情况，写为0或者1。如果是两个比特，那就有四种可能的组合，00、01、10和11。通常，如果你有 `b` 个比特，你就可以表示 `2 ** b` 个值之一。一个字节是8个比特，所以它可以储存256个值之一。

从其它方面来讲，假设你想要储存字母表中的字母。字母共有26个，所以你需要多少个比特呢？使用4个比特你可以表示16个值之一，这是不够的。使用5个比特你可以表示32个值，这对于所有字母是够用的，同时还有一点点浪费。

通常，如果你想要表示 `N` 个值之一，你就需要求出最小的 `b` 使 `2 ** b >= N` 。在两边计算以2为底的对数，就会得到 `b >= log(2, N)` 。

假设我投掷一枚硬币并且告诉你结果，我就向你提供了1比特的信息。如果我投掷六个面的筛子并告诉你结果，我就向你提供了 `log(2, 6)` 比特的信息。并且通常，如果结果的概率是 `1/n` ，结果应该包含 `log(2, N)` 比特的信息。

同样，如果结果的概率为 `p` ，那么信息的内容为 `-log(2, p)` 。这个数量叫做“自信息”（self-information）。它度量了结果有多么令人意外，所以也叫作“惊异度”。如果你的赛马只有十六分之一的几率获胜，并且它获胜了，那么你就得到了4比特的信息（以及奖金）。但是如果它的获胜几率为75%，这条新闻只含有0.42个比特。

可以由直觉得出，非预期的新闻会带有大量信息；与之相反，如果你对一件事情很有自信，对它的验证只会得到少量的信息。

对于书中的一些话题，我们只需要熟练于在比特数量 `b` 和它们所编码的值的数量 `N = 2 ** b` 之间进行转换。

3.2 内存（Memory）和储存器（Storage）

当进程处于运行期间，它的多数数据都放在“主存”（内存）之中，它通常是一些随机储存器（RAM）。在当前的大多数电脑上，主存非常易失，也就是说，当电脑关闭时，主存的内容就没了。一个典型的台式电脑拥有2~8GiB的内存。GiB代表“gibibyte”，相当于 `2 ** 30` 个字节。

如果进程会读写文件，这些文件通常放在机械硬盘（HDD）或固态硬盘（SSD）里面。这些储存器都是非易失的，所以他们可用于长时间储存。当前，一个典型的台式电脑拥有500GB到2TB的HDD。GB代表“gigabyte”，相当于 `10 ** 9` 个字节。TB代表“terabyte”，相当于 `10 ** 12` 个字节。

你可能会注意到我使用二进制单位GiB来描述主存大小，并使用十进制单位GB和TB来描述HDD的大小。由于历史和技术因素，内存以二进制单位度量，并且硬盘以十进制单位度量。本书中我会小心区分二进制和十进制单位，但是你应该注意到“gigabyte”以及GB缩写通常在使用上非常模糊。

非正式的用法中，“内存”有时会用于HDD和SSD（特别是移动设备），以及RAM。然而，这些设备的属性大相径庭，所以我们需要区分它们。我会使用“储存器”来指代HDD和SSD。

3.3 地址空间

主存中的每个字节都由一个“物理地址”整数所指定，物理地址的集合叫做物理“地址空间”。它的范围通常为0到 `N-1` ，其中 `N` 是主存的大小。在带有1GiB主存的的系统上，最高的有效地址是 `2 ** 30 - 1` ，十进制表示为1,073,741,823，16进制表示为0x03ff ffff（前缀 `0x` 表示十六进制）。

然而，许多操作系统提供“虚拟内存”，也就是说程序永远不需要处理物理地址，也不需要知道有多少物理内存是有效的。

作为代替，程序处理虚拟地址，它被编码为从0到 `M-1` ，其中 `M` 是有效虚拟地址的大小。虚拟地址空间的大小取决于所处的操作系统和硬件。

你一定听过人们谈论32位和64位系统。这些术语表明了寄存器的尺寸，也通常是虚拟地址的大小。在32位系统上，虚拟地址是32位的，也就是说虚拟地址空间为从0到0xffff ffff。这一地址空间的大小是 `2 ** 32` 个字节，或者4GiB。

在64位系统上，虚拟地址空间大小为 `2 ** 64` 个字节，或者 `4 * 1024 ** 6` 个字节。这是16个EiB，大约比当前的物理内存大十亿倍。虚拟内存比物理内存大很多，这看上去有些奇怪，但是我们很快就就会看到它如何工作。

当一个程序读写内存中的值时，它使用虚拟地址。硬件在操作系统的帮助下，在访问主存之前将物理地址翻译成虚拟地址。翻译过程在进程层级上完成，所以即使两个进程访问相同的虚拟地址，它们所映射的物理地址可能不同。

因此，虚拟内存是操作系统隔离进程的一种重要途径。通常，一个进程不能访问其他进程的数据，因为没有任何虚拟地址能映射到其他进程分配的物理内存。

3.4 内存段

一个运行中进程的数据组织为4个段：

- `text` 段包含程序文本，即程序所组成的机器语言指令、
- `static` 段包含由编译器所分配的变量，包括全局变量，和使用 `static` 声明的局部变量。
- `stack` 段包含运行时栈，它由栈帧组成。每个栈帧包含函数参数、本地变量以及其它。
- `heap` 段包含运行时分配的内存块，通常通过调用C标准库函数 `malloc` 来分配。

这些段的组织方式部分取决于编译器，部分取决于操作系统。不同的操作系统中细节可能不同，但是下面这些是共同的：

- `text` 段靠近内存“底部”，即接近0的地址。
- `static` 段通常刚好在 `text` 段上面。
- `stack` 段靠近内存顶部，即接近虚拟地址空间的最大地址。在扩张过程中，它向低地址的方向增长。
- `heap` 通常在 `static` 段的上面。在扩张过程中，它向高地址的方向增长。

为了搞清楚这些段在你操作系统上的布局，可以尝试运行这个程序，它就是这本书的仓库中的 `aspace.c` ：

```
#include <stdio.h>
#include <stdlib.h>

int global;

int main ()
{
    int local = 5;
    void *p = malloc(128);

    printf ("Address of main is %p\n", main);
    printf ("Address of global is %p\n", &global);
    printf ("Address of local is %p\n", &local);
    printf ("Address of p is %p\n", p);
}
```

`main` 是函数的名称，当它用作变量时，它指向 `main` 中第一条机器语言指令的地址，我们认为它在 `text` 段内。

`global` 是一个全局变量，所以我们认为它在 `static` 段内。 `local` 是一个局部变量，所以我们认为它在栈上。

`p` 持有 `malloc` 所返回的地址，它指向堆区所分配的空间。 `malloc` 代表“内存分配”（memory allocate）。

格式化占位符 `%p` 告诉 `printf` 把每个地址格式化为“指针”，它是地址的另一个名字。

当我运行这个程序时，输出就像下面这样（我添加了空格使它更加易读）：

```
Address of main is   0x       40057c
Address of global is 0x       60104c
Address of local is  0x7fffd26139c4
Address of p is      0x      1c3b010
```

正如预期的那样，`main` 的地址最低，随后是 `global` 和 `p` 。 `local` 的地址会更大，它是12个十六进制数字，每个十六进制数字对应4比特，所以它是48位的地址。这表明虚拟内存的可用部分为 `2 ** 48` 个字节。

作为一个练习，你需要在你的电脑上运行这个程序，并将你的结果与我的结果比较。添加对 `malloc` 的第二个调用来检查你系统上的堆区是否向上增长（地址更高）。添加一个函数来打印出局部变量的地址，检查栈是否向下增长。

3.5 静态局部变量

栈上的局部变量有时称为“自动变量”，因为它们当函数创建时自动被分配，并且当函数返回时自动被释放。

C语言中又另一种局部变量，叫做“静态变量”，它分配在在 `static` 段上。它在程序启动时初始化，并且在函数调用之间保存它的值。

例如，下面的函数跟踪了它所调用的次数：

```
int times_called()
{
  static int counter = 0;
  counter++;
  return counter;
}
```

`static` 关键字表示 `counter` 是静态局部变量。它的初始化只发生一次，就是程序启动的时候。

如果你将这个函数添加到 `aspace.c` ，你可以确定 `counter` 和全局变量一起分配在 `static` 段上，而不是在栈上。

3.6 地址翻译

虚拟地址（VA）如何翻译成物理地址（PA）？基本的机制十分简单，但是简单的实现方式十分耗时，并且占据大量空间。所以实际的实现会复杂一点。

大多数处理器提供了内存管理单元（MMU），位于CPU和主存之间。MMU在VA和PA之间执行快速的翻译。

1. 当程序读写变量时，CPU会得到VA。
2. MMU将VA分成两部分，称为页码和偏移。“页”是一个内存块，页的大小取决于操作系统和硬件，通常为1~4KiB。
3. MMU在“页表”里查找页码，然后获取相应的物理页码。之后它将物理页码和偏移组合得到PA。
4. PA传递给主存，用于读写指定地址。

作为一个例子，假设VA为32位，物理内存为1GiB，划分为1KiB的页面。

- 由于1GiB为 `2 ** 30` 个字节，物理页的数量为 `2 ** 20` 个，它们也称为“帧”。
- 虚拟地址空间的大小为 `2 ** 32` 字节，这个例子中，页的大小为 `2 ** 10` 字节，所以共有 `2 ** 22` 个虚拟页。
- 偏移的大小取决于页的大小。这个例子中页的大小为 `2 ** 10` 字节，所以需要10位来指定页中的一个字节。
- 如果VA是32位，而偏移是10位，剩余的22位构成了虚拟页码。
- 由于共有 `2 ** 20` 个物理页，每个物理页码是20位。加上10位的偏移，PA的结果为30位。

到目前为止，看上去是可行的。但是让我们考虑一下页表应该占多大。页表最简单的实现是一个数组，每个虚拟页面是一个条目。每个条目都包含一个物理页码，在例子中它是20位，加上每帧的一些额外的数据，所以我们认为每个条目占用3~4个字节。由于共有 `2 ** 22` 个虚拟页，页面共需要 `2 ** 24` 个字节，或16MiB。

由于我们需要为每个进程创建一个页表，一个运行256个进程的系统就需要 `2 ** 32` 个字节，或者4GiB，这还只是页表的空间！这些就占用了全部32位虚拟地址。而在48或64位的虚拟地址上，这个数量更加荒谬。

幸运的是，并不需要这么大的空间，因为大多数进程不使用虚拟地址空间的每个小片段。而且，如果一个进程不使用某个虚拟页面，我们也不需要在页表中为其分配条目。

也就是说，页表是“稀疏”的，这暗示了最简单的实现，即页表条目的数组是个糟糕的想法。幸运的是，稀疏数组有一些不错的实现方式。

一种选择是多级页表，它被多数操作系统例如Linux所采用。另一种选择是关联表，其中每个条目包含虚拟页码和物理页码。在软件上搜索关联表会非常慢，但是硬件上我们可以并行搜索整个表，所以关联数组经常用于在MMU中表示页表。

你可以在页表的维基百科页面阅读更多关于这些实现的信息。你也可能会找到有趣的细节。但是基本的想法就是页表应做成稀疏的，所以我们需要为稀疏数组选择一个好的实现方式。

我之前提到了操作系统可以中断一个运行中的进程，保存它的状态，之后运行其它进程。这个机制叫做“上下文切换”。由于每个进程都有自己的页表，操作系统需要和MMU配合来保证每个进程拿到了正确的页表。在旧机器上，MMU中的页表信息在每次上下文切换时会被替换掉，开销非常大。在新的系统中，MMU的每个页表条目包含进程ID，所以多个进程的页表可以同时储存在MMU中。

第四章 文件和文件系统

> 作者：Allen B. Downey
>
> 原文：Chapter 4 Files and file systems
>
> 译者：飞龙
>
>

当一个进程运行完毕（或崩溃）时，任何储存在主存的数据都会丢失。但是储存在机械硬盘（HDD）或固态硬盘（SSD）的数据是“持久”的。也就是说，它在进程结束之后，甚至关机之后仍旧存在。

机械硬盘比较复杂。数据存储在块内，它们布局在扇区中，扇区又组成磁道。磁道在盘片上以同心圆的形式排列。

固态硬盘稍微简单一些，因为块按顺序被标号。但是这会产生另一种困难，每个块在变得不可靠之前，只能被读写有限的次数。

作为一个程序员，你并不需要处理这些难题。你只需要硬件持久化存储的一个适当抽象。最普遍的抽象叫做“文件系统”。

在抽象上：

- “文件系统”将每个文件的名称映射到它的内容。如果你认为名称是键，内容是值，文件系统就是一种键值对的数据库。
- “文件”就是一组字节序列。

文件名通常是字符串，并且通常是分层的。这就是说，这个字符串指定了顶级目录（或文件夹）的路径，通过一系列子目录，到达特定的文件。

这个抽象和底层机制的根本不同，就是文件是基于字节的，而持久化储存器是基于块的。操作系统将C标准库中基于字节的文件操作翻译成基于块的储存设备操作。每个块的典型大小是1~8KiB。

例如，下面的代码打开文件并读取首个字节：

```
FILE *fp = fopen("/home/downey/file.txt", "r");
char c = fgetc(fp);
fclose(fp);
```

当这段代码执行时：

1. `fopen`使用文件名来寻找顶级目录，叫做`/`，子目录`/home`，和二级子目录`downey`。
2. 它找到了名为`file.txt`的文件，并且“打开”它以便读取。意思是它创建了一个数据结构

来表示将要读取的文件。除此之外，这个数据结构还跟踪了文件读取了多少字节，称为“文件位置”。
3. 当我们调用 `fgetc` 时，操作系统检查下个字节是否已经在内存里了。如果是的话，它会读取下一个字节，向前移动文件位置，并返回结果、
4. 如果下一个字节不在内存中，操作系统产生IO请求来获取下一个块。硬盘非常慢，所以一个等待磁盘块的进程通常会被中断，直到数据到达之前，都在运行另一个进程。
5. IO操作完成时，新的数据块会储存在内存中，进程也会恢复运行。它读取第一个字节并把它储存在局部变量中。
6. 当进程关闭文件时，操作系统完成或取消任何等待中的操作，移除内存中的数据，并且释放 `OpenFileTableEntry` 。

写入文件的过程与之相似，但是有一些额外的步骤。下面是一个例子，打开文件用于读取，并且修改首个字节：

```
FILE *fp = fopen("/home/downey/file.txt", "w");
fputc('b', fp);
fclose(fp);
```

当这段代码执行时：

1. 同样， `fopen` 使用文件名来寻找文件。如果它不存在，就会创建新的文件，并向父目录 `/home/downey` 添加条目。
2. 操作系统创建 `OpenFileTableEntry` ，表示这个文件已打开等待写入，并将文件位置设置为0。
3. `fputc` 尝试写入（或者覆写）文件的第一个字节。如果文件已经存在，操作系统需要将第一个块加载到内存中。否则它会在内存中分配新的块，并且在磁盘上请求新的块。
4. 在内存中的块被修改之后，可能不会立即复制回磁盘。通常，写到文件中的数据是“被缓冲的”，意思是它储存在内存中，只在至少有一个块需要写入时才写回磁盘。
5. 文件关闭时，任何缓冲的数据都会写到磁盘，并且 `OpenFileTableEntry` 会被释放。

总之，C标准库提供了文件系统的抽象，将文件名称映射到字节流。这个抽象建立在实际以块组织的储存设备之上。

4.1 磁盘性能

我之前提到过，磁盘驱动器非常慢。在当前的HDD上，从磁盘读取一个块到内存的时间为2~6毫秒。SSD要快一些，读取4KiB的块需要25微秒，写入需要250微秒（请见 http://en.wikipedia.org/wiki/Ssd#Controller）。

为了正确看待这些数据，让我们将其与CPU的时钟周期进行比较。一个拥有2GHZ时钟频率的处理器，每0.5纳秒就会完成一个时钟周期。从内存获取一个字节到CPU的时间通常为100纳秒。如果处理器每个时钟周期完成一条指令，在等待来自内存的一个字节时，它可以完成200条指令。

在一微秒内，它可以完成2000条指令，所以在等待来自SSD的一个字节时，它可以完成50000条。

在一毫秒内，它可以完成2,000,000条指令，所以在等待来自HDD的一个字节时，它可以完成一千万条。如果CPU在等待期间没有什么事情要做，就会闲置。这就是操作系统在等待来自磁盘的数据时，通常会切换到另一个进程的原因。

主存和持久化储存器的性能间隔是计算机系统的主要挑战之一。操作系统和硬件提供了一些特性来“填补”这一间隔。

- 块的传输：从磁盘加载一个字节的时间是5毫秒。相比之下，加载一个8KiB的块所需的时间是微不足道的。如果处理器在每个块上都要花费5毫秒，就有可能使处理器保持忙碌。
- 预取：有时操作系统可以预测到进程会读取某个块，并且在它请求之前就开始加载了。例如，如果你打开一个文件并读取首个块，操作系统可能会在请求之前开始加载额外的块。
- 缓冲：像我提到过的那样，当你写入一个文件时，操作系统会先把数据放在内存中，并且稍后写到磁盘。如果某个块在内存中时，你对其做数次修改，系统只需要写到磁盘一次。

这些特性中一部分实现在硬件上。例如，一些硬盘驱动器提供了缓存功能来储存最近所使用的块。许多磁盘驱动器也会一次读取多个块，即使只请求了一个块。

这些机制通常改进了程序的性能，但是它们并不改变行为。通常程序员不需要考虑它们，除了两个例外：（1）如果程序的性能十分差劲，你可能需要了解这些机制来判断问题所在。或者（2）当数据被缓冲时，调试程序就变得很困难。例如，如果程序打印出一个值，然后崩溃。这个值就可能不会出现，因为它可能位于缓冲区中。与此相似，如果一个程序向磁盘写入数据，之后计算机没电了。如果数据位于缓存中，还没有写到磁盘，就可能会丢失。

4.2 磁盘元数据

组成文件的块可能在磁盘上是连续排列的，如果它们是这样，文件系统的性能会高一些。但是大多数操作系统并不需要连续的分配，它们可以将某个块放在磁盘上的任意位置，并且使用各种数据结构来跟踪这些块。

在许多Unix文件系统中，这些数据结构叫做`inode`，它代表“索引节点”（index node）。更通常来说，关于文件的信息，包括所包含的块的位置，叫做“元数据”。（文件内容就是数据，所以关于文件内容的数据就是数据的数据，所以为“元数据”。）

由于inode和其余数据一样位于磁盘上，它们被设计来巧妙地整合进磁盘块中。Unix的inode包含关于文件的信息，这包括：文件拥有者的用户ID，表明谁可以读写或执行的权限位，以及表明最后修改和访问时间的时间戳。另外，对于组成文件的前12个块，它包含了块的数量。

如果每个块的大小是8KiB，前12个块合计96KiB。在大多数系统中，这对于大多数文件就足够了，但是，这对于所有文件明显不一定够用。这就是inode同时也包含一个指向“间接块”指针的原因，间接块包含了指向其它块的指针。

间接块的指针数量取决于块的数量和大小，它通常是1024。如果有1024个块，每个块是8KiB，那么一个间接块可以编址8MiB。这对于大多数大文件就够了，但对于所有大文件还是不够。

这就是inode同时含有“二级间接块”指针的原因，二级间接块含有指向间接块的指针。我们可以使用1024个间接块来编址8GiB。

如果这样还是不够大，最后有一个三级间接块，它含有指向二级间接块指针，支持最大8TiB的文件大小。Unix的inode在设计时，它似乎在很长一段时间内都是够大的。但是那是很久之前了。

作为间接块的替代，一些文件系统，例如FAT，使用了一张文件分配表，它为每个块包含一个条目，在这个上下文中叫做“簇”。根目录包含指向每个文件第一个簇的指针。FAT上每个簇的条目指向文件中的下一个簇，就像链表那样。更多请见文件分配表的维基百科。

4.3 块的分配

操作系统需要跟踪哪些块属于每个文件，它们也需要跟踪哪些块可供使用。当新的文件创建时，文件系统会寻找可用的块并且分配它。当文件删除时，文件系统会释放它的块用于再次分配。

块分配系统的目标是：

- 速度：块的分配和释放应该很快。
- 最小的空间开销：用于分配器的数据结构应尽可能小，把尽可能多的空间留给数据。
- 最少的碎片：如果一些块没有被使用，或者只是部分使用，没有使用的空间被称为“碎片”。
- 最大的连续性：可能同时使用的数据应尽可能物理连续，以便提高性能。

设计一个满足以上所有目标的文件系统很困难，尤其是由于文件系统的性能取决于“工作负载的特征”，包括文件大小、访问模式以及其它。对于某种工作负载表现良好的文件系统，可能对于其它工作负载的表现并不好。

由于这种因素，大多数操作系统支持多种文件系统，并且文件系统的设计是一个活跃的研究和发展领域。近十年中，Linux系统由ext2迁移到ext3。前者是一种传统的Unix文件系统，而后者是一种用于提高速度和连续性的日志文件系统。最近它迁移到了ext4，它可以处理更大的文件和文件系统。在几年之内，可能又会迁移到基于B树的文件系统，Btrfs。

4.4 任何东西都是文件吗？

文件抽象实际上是“字节流”的抽象，这对于很多事情都很实用，不仅仅是文件系统。

一个例子是Unix管道，它是进程间通信的一个简单形式。可以建立这样一些进程，使一个进程的输出用作另一个进程的输入。对于第一个进程，管道表现为打开用于写入的文件，所以它可以使用C标准库类似 `fputs` 和 `fprintf` 的函数。对于第二个进程，管道表现为打开用于读取的文件，所以它可以使用 `fgets` 和 `fscanf` 。

网络通信也使用了字节流的抽象。Unix套接字是一个数据结构，它（通常）表示两个不同电脑上的进程之间的信道。同样，进程可以使用“文件”处理函数从套接字读取数据和向套接字写入数据。

复用文件抽象使程序员的工作变得容易，因为他们只需要了解一套API（应用程序接口）。这也使程序具有多种功能，因为一个需要处理文件的程序还可以处理来自管道和其它来源的数据。

第五章 更多的位与字节

> 作者：Allen B. Downey
>
> 原文：Chapter 5 More bits and bytes
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0

5.1 整数的表示

你可能知道计算机以二进制表示整数。对于正数，二进制的表示法非常直接。例如，十进制的5表示成二进制是`0b101`。

对于负数，最清晰的表示法使用符号位来表明一个数是正数还是负数。但是还有另一种表示法，叫做“补码”（two's complement），它更加普遍，因为它和硬件配合得更好。

为了寻找一个负数`-x`的补码，需要找到`x`的二进制表示，将所有位反转，之后加上1。例如，要表示`-5`（十进制），要先从5（十进制）开始，如果将其写成8位的形式它是`0b0000 0101`。将所有位反转并加1会得到`0b1111 1011`。

在补码中，最左边的位相当于符号位。正数中它是0，负数中它是1。

为了将8位的数值转换为16位，我们需要对正数添加更多的0，对负数添加更多的1。实际上，我们需要将符号位复制到新的位上，这个过程叫做“符号扩展”。

在C语言中，除非你用`unsigned`声明它们，所有整数类型都是有符号的（能够表示正数和负数）。它们之间的差异，以及这个声明如此重要的原因，是无符号整数上的操作不使用符号扩展。

5.2 按位运算

学习C语言的人有时会对按位运算`&`和`|`感到困惑。这些运算符将整数看做位的向量，并且在相应的位上执行逻辑运算。

例如，`&`执行“且”运算。如果两个操作数都为1结果为1，否则为0。下面是一个在两个4位数值上执行`&`运算的例子：

```
  1100
& 1010
  ----
  1000
```

C语言中，这意味着表达式`12 & 10`值为8。

与之相似，`|`执行“或”运算，如果两个操作数至少一个为1结果为1，否则为0。

```
  1100
| 1010
  ----
  1110
```

所以表达式`12 | 10`值为14。

最后，`^`运算符执行“异或”运算，如果两个操作数有且仅有一个为1，结果为1。

```
  1100
^ 1010
  ----
  0110
```

所以表达式`12 ^ 10`值为6。

通常，`&`用于清除位向量中的一些位，`|`用于设置位，`^`用于反转位。下面是一些细节：

清除位：对于任何`x`，`x & 0`值为0，`x & 1`值为`x`。所以如果你将一个向量和3做且运算，它只会保留最右边的两位，其余位都置为0。

```
  xxxx
& 0011
  ----
  00xx
```

在这个语境中，3叫做“掩码”，因为它选择了一些位，并屏蔽了其余的位。

设置位：与之相似，对于任何`x`，`x | 0`值为`x`，`x | 1`值为1。所以如果你将一个向量与3做或运算，它会设置右边两位，其余位不变。

```
  xxxx
| 0011
  ----
  xx11
```

反转位：最后，如果你将一个向量与3做异或运算，它会反转右边两位，其余位不变。作为一个练习，看看你能否使用`^`计算出12的补码。提示：-1的补码表示是什么？

C语言同时提供了移位运算符，`<<`和`>>`，它可以将位向左或向右移。向左每移动一位会使数值加倍，所以`5 << 1`为10，`5 << 2`为20。向右每移动一位会使数值减半（向下取整），所以`5 >> 1`为2，`2 >> 1`为1。

5.3 浮点数的表示

浮点数使用科学计数法的二进制形式来表示。在十进制的形式中，较大的数字写成系数与十的指数相乘的形式。例如，光速大约是 `2.998 * 10 ** 8` 米每秒。

大多数计算机使用IEEE标准来执行浮点数运算。C语言的 `float` 类型通常对应32位的IEEE标准，而 `double` 通常对应64位的标准。

在32位的标准中，最左边那位是符号位， `s` 。接下来的8位是指数 `q` ，最后的23位是系数 `c` 。浮点数的值为：

```
(-1) ** s * c * 2 ** q
```

这几乎是正确的，但是有一点例外。浮点数通常为规格化的，所以小数点前方有一个数字。例如在10进制中，我们通常使用 `2.998 * 10 ** 8` 而不是 `2998 * 10 ** 5` ，或者任何其它等价的表示。在二进制中，规格化的浮点数总是在二进制小数点前有一个数字1。由于这个位置上的数字永远是1，我们可以将其从表示中去掉以节省空间。

例如，十进制的13表示为 `0b1101` ，在浮点数中，它就是 `1.101 * 2 ** 3` 。所以指数为3，系数储存为101（加上20个零）。

这几乎是正确的，但是指数以“偏移”储存。在32位的标准中，偏移是127，所以指数3应该储存为130。

为了在C中对浮点数打包和解包，我们可以使用联合体和按位运算，下面是一个例子：

```
union {
    float f;
    unsigned int u;
} p;

p.f = -13.0;
unsigned int sign = (p.u >> 31) & 1;
unsigned int exp = (p.u >> 23) & 0xff;

unsigned int coef_mask = (1 << 23) - 1;
unsigned int coef = p.u & coef_mask;

printf("%d\n", sign);
printf("%d\n", exp);
printf("0x%x\n", coef);
```

这段代码位于这本书的仓库的 `float.c` 中。

联合体可以让我们使用 `p.f` 储存浮点数，之后将使用 `p.u` 当做无符号整数来读取。

为了获取符号位，我们需要将其右移31位，之后使用1位的掩码选择最右边的位。

为了获取指数，我们需要将其右移23位，之后选择最右边的8位（十六进制值 `0xff` 含有8个1）。

为了获取系数，我们需要解压最右边的23位，并且忽略掉其余位，通过构造右边23位是1并且其余位是0的掩码。最简单的方式是将1左移23位之后减1。

程序的输出如下：

```
1
130
0x500000
```

就像预期的那样，负数的符号位为1。指数是130，包含了偏移。而且系数是101带有20个零，我用十六进制将其打印了出来。

作为一个练习，尝试组装或分解 `double` ，它使用了64位的标准。请见IEEE浮点数的维基百科。

5.4 联合体和内存错误

C的联合体有两个常见的用处。一个是就是在上一节看到的那样，用于访问数据的二进制表示。另一个是储存不同形式的数据。例如，你可以使用联合体来表示一个可能为整数、浮点、复数或有理数的数值。

然而，联合体是易于出错的，这完全取决于你，作为一个程序员，需要跟踪联合体中的数据类型。如果你写入了浮点数然后将其读取为整数，结果通常是无意义的。

实际上，如果你错误地读取内存的某个位置，也会发生相同的事情。其中一种可能的方式是越过数组的尾部来读取。

我会以这个函数作为开始来观察所发生的事情。这个函数在栈上分配了一个数组，并且以0到99填充它。

```
void f1() {
    int i;
    int array[100];

    for (i=0; i<100; i++) {
        array[i] = i;
    }
}
```

接下来我会定义一个创建小型数组的函数，并且故意访问在开头之前和末尾之后的元素：

```
void f2() {
    int x = 17;
    int array[10];
    int y = 123;

    printf("%d\n", array[-2]);
    printf("%d\n", array[-1]);
    printf("%d\n", array[10]);
    printf("%d\n", array[11]);
}
```

如果我一次调用 `f1` 和 `f2` ，结果如下：

```
17
123
98
99
```

这里的细节取决于编译器，它会在栈上排列变量。从这些结果中我们可以推断，编译器将`x`和`y`放置到一起，并位于数组“下方”（低地址处）。当我们越过数组的边界读取时，似乎我们获得了上一个函数调用遗留在栈上的数据。

这个例子中，所有变量都是整数，所以比较容易弄清楚其原理。但是通常当你对数组越界读取时，你可能会读到任何类型的值。例如，如果我修改`f1`来创建浮点数组，结果就是：

```
17
123
1120141312
1120272384
```

最后两个数值就是你将浮点数解释为整数的结果。如果你在调试时遇到这种输出，你就很难弄清楚发生了什么。

5.5 字符串的表示

字符串有时也会有相关的问题。首先，要记住C的字符串是以空字符结尾的。当你为字符串分配空间时，不要忘了末尾额外的字节。

同样，要记住C字符串中的字母和数字都编码为ASCII码。数字0~9的ASCII码是48~57，而不是0~9。ASCII码的0是`NUL`字符，用于标记字符串的末尾。ASCII码的1~9是用于一些通信协议的特殊字符。ASCII码的7是响铃，在一些终端中，打印它们会发出声音。

`'A'`的ASCII码是65，`'a'`是97，下面是它们的二进制形式：

```
65 = b0100 0001
97 = b0110 0001
```

细心的读者会发现，它们只有一位的不同。这个规律对于其余所有字符都适用。从右数第六位起到“大小写”位的作用，0表示大写字母，1表示小写字母。

作为一个练习，编写一个函数，接收字符串并通过反转第六位将小写字符转换成大写字母。作为一个挑战，你可以通过一次读取字符串的32位或64位而不是一个字符使它更快。如果字符串的长度是4或8字节的倍数，这个优化会容易实现一些。

如果你越过字符串的末尾来读取，你可能会看到奇怪的字符。反之，如果你创建了一个字符串，之后无意中将其作为整数或浮点读取，结果也难以解释。

例如，如果你运行：

```
char array[] = "allen";
float *p = array;
printf("%f\n", *p);
```

你会发现我的名字的前8个字符的ASCII表示，可以解释为一个双精度的浮点，它是69779713878800585457664。

第六章 内存管理

> 作者：Allen B. Downey
>
> 原文：Chapter 6 Memory management
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0

C提供了4种用于动态内存分配的函数：

- `malloc`，它接受表示字节单位的大小的整数，返回指向新分配的、（至少）为指定大小的内存块的指针。如果不能满足要求，它会返回特殊的值为`NULL`的指针。
- `calloc`，它和`malloc`一样，除了它会清空新分配的空间。也就是说，它会设置块中所有字节为0。
- `free`，它接受指向之前分配的内存块的指针，并会释放它。也就是说，使这块空间可用于未来的分配。
- `realloc`，它接受指向之前分配的内存块的指针，和一个新的大小。它使用新的大小来分配内存块，将旧内存块中的数据复制到新内存块中，释放旧内存块，并返回指向新内存块的指针。

这套API是出了名的易错和苛刻。内存管理是设计大型系统中，最具有挑战性的一部分，它正是许多现代语言提供高阶内存管理特性，例如垃圾回收的原因。

6.1 内存错误

C的内存管理API有点像Jasper Beardly，动画片《辛普森一家》中的一个配角，他是一个严厉的代课老师，喜欢体罚别人，并使用戒尺惩罚任何违规行为。

下面是一些应受到惩罚的程序行为：

- 如果你访问任何没有分配的内存块，就应受到惩罚。
- 如果你释放了某个内存块之后再访问它，就应受到惩罚。
- 如果你尝试释放一个没有分配的内存块，就应受到惩罚。
- 如果你释放多次相同的内存块，就应受到惩罚。
- 如果你使用没有分配或者已经释放的内存块调用`realloc`，就应受到惩罚。

这些规则听起来好像不难遵循，但是在一个大型程序中，一块内存可能由程序一部分分配，在另一个部分中使用，之后在其他部分中释放。所以一部分中的变化也需要其它部分跟着变化。

同时，同一个内存块在程序的不同部分中，也可能有许多别名或者引用。这些内存块在所有引用不再使用时，才应该被释放。正确处理这件事情通常需要细心的分析程序的所有部分，这非常困难，并且与良好的软件工程的基本原则相违背。

理论上，每个分配内存的函数都应包含内存如何释放的信息，作为接口文档的一部分。成熟的库通常做得很好，但是实际上，软件工程的实践通常不是这样理想化的。

内存错误非常难以发现，因为这些症状是不可预测的，这使得事情更加糟糕，例如：

- 如果从未分配的内存块中读取值，系统可能会检测到错误，触发叫做“段错误”的运行时错误，并且中止程序。这个结果非常合理，因为它表示程序所读取的位置会导致错误。但是，遗憾的是，这种结果非常少见。更通常的是，程序读取了未分配的内存块，而没有检测到错误，程序所读取的未分配内存正好储存在一块特定区域中。如果这个值没有解释为正确的类型，结果可能会难以解释。例如，如果你读取字符串中的字节，将它们解释为浮点数，你可能会得到一个无效的数值，非常大或非常小的数值。如果你向函数传递它无法处理的值，结果会非常怪异。
- 如果你向未分配的内存块中写入值，会更加糟糕。因为在值被写入之后，需要很长时间值才能被读取并且发生错误。此时寻找问题来源就会非常困难。事情还可能更加糟糕！C风格内存管理的一个最普遍的问题是，用于实现 `malloc` 和 `free` 的数据结构（我们将会看到）通常和分配的内存块储存在一起。所以如果你无意中越过动态分配块的末尾写入值，你就可能破坏了这些数据结构。系统通常直到最后才会检测到这种问题，当你调用 `malloc` 或 `free` 时，这些函数会由于一些谜之原因调用失败。

你应该从中总结出一条规律，就是安全的内存管理需要设计和规范。如果你编写了一个分配内存的库或模块，你应该同时提供释放它的接口，并且内存管理从开始就应该作为API设计的一部分。

如果你使用了分配内存的库，你应该按照规范使用API。例如，如果库提供了分配和释放储存空间的函数，你应该一起使用或都不使用它们。例如，不要在不是 `malloc` 分配的内存块上调用 `free` 。你应该避免在程序的不同部分中持有相同内存块的多个引用。

通常在安全的内存管理和性能之间有个权衡。例如，内存错误的的最普遍来源是数组的越界写入。这一问题的最显然的解决方法就是边界检查。也就是说，每次对数组的访问都应该检查下标是否越界。提供数组结构的高阶库通常会进行边界检查。但是C风格数据和大多数底层库不会这样做。

6.2 内存泄漏

有一种可能会也可能不会受到惩罚的内存错误。如果你分配了一块内存，并且没有释放它，就会产生“内存泄漏”。

对于一些程序，内存泄露是OK的。如果你的程序分配内存，对其执行计算，之后退出，这可能就不需要释放内存。当程序退出时，所有分配的内存都会由操作系统释放。在退出前立即释放内存似乎很负责任，但是通常很浪费时间。

但是如果一个程序运行了很长时间，并且泄露内存的话，它的内存总量会无限增长。此时会发生一些事情：

- 某个时候，系统会耗完所有物理内存。在没有虚拟内存的系统上，下一次的 `malloc` 调用会失败，返回 `NULL` 。
- 在带有虚拟内存的系统上，操作系统可以将其它进程的页面从内存移动到磁盘上，之后分配更多空间给泄露的进程。我会在7.8节解释这一机制。
- 单个进程可能有内存总量的限制，超过它的话， `malloc` 会返回 `NULL` 。
- 最后，进程可能会用完它的虚拟地址空间（或者可用的部分）。之后，没有更多的地址可分配， `malloc` 会返回 `NULL` 。

如果 `malloc` 返回了 `NULL` ，但是你仍旧把它当成分配的内存块进行访问，你会得到段错误。因此，在使用之前检查 `malloc` 的结果是个很好的习惯。一种选择是在每个 `malloc` 调用之后添加一个条件判断，就像这样：

```
void *p = malloc(size);
if (p == NULL) {
    perror("malloc failed");
    exit(-1);
}
```

`perror` 在 `stdio.h` 中声明，它会打印出关于最后发生的错误的错误信息和额外的信息。

`exit` 在 `stdlib.h` 中声明，会使进程终止。它的参数是一个表示进程如何终止的状态码。按照惯例，状态码0表示通常终止，-1表示错误情况。有时其它状态码用于表示不同的错误情况。

错误检查的代码十分讨厌，并且使程序难以阅读。但是你可以通过将库函数的调用和错误检查包装在你自己的函数中，来解决这个问题。例如，下面是检查返回值的 `malloc` 包装：

```
void *check_malloc(int size)
{
  void *p = malloc (size);
  if (p == NULL) {
    perror("malloc failed");
    exit(-1);
  }
  return p;
}
```

由于内存管理非常困难，多数大型程序，例如Web浏览器都会泄露内存。你可以使用Unix的 `ps` 和 `top` 工具来查看系统上的哪个程序占用了最多的内存。

6.3 实现

当进程启动时，系统为 `text` 段、静态分配的数据、栈和堆分配空间，堆中含有动态分配的数据。

并不是所有程序都动态分配数据，所以堆的大小可能很小，或者为0。最开始堆只含有一个空闲块。

`malloc` 调用时，它会检查这个空闲块是否足够大。如果不是，它会向系统请求更多内存。做这件事的函数叫做 `sbrk` ，它设置“程序中断点”（program break），你可以将其看做一个指向堆底部的指针。

> 译者注： `sbrk` 是Linux上的系统API，Windows上使用 `HeapAlloc` 和 `HeapFree` 来管理堆区。

`sbrk` 调用时，它分配的新的物理内存页，更新进程的页表，并设置程序中断点。

理论上，程序应该直接调用 `sbrk` （而不是通过 `malloc` ），并且自己管理堆区。但是 `malloc` 易于使用，并且对于大多数内存使用模式，它运行速度快并且高效利用内存。

为了实现内存管理API，多数Linux系统都使用 `ptmalloc` ，它基于 `dlmalloc` ，由Doug Lea编写。一篇描述这个实现要素的论文可在http://gee.cs.oswego.edu/dl/html/malloc.html访问。

对于程序员来说，需要注意的最重要的要素是：

- `malloc` 在运行时通常不依赖块的大小，但是可能取决于空闲块的数量。 `free` 通常很快，和空闲块的数量无关。因为 `calloc` 会清空块中的每个字节，执行时间取决于块的大小（以及空闲块的数量）。 `realloc` 有时很快，如果新的大小比之前更小，或者空间可用于扩展现有的内存块。否则，它需要从旧内存块中复制数据到新内存块，这种情况下，执行时间取决于旧内存块的大小。
- 边界标签：当 `malloc` 分配一个快时，它在头部和尾部添加空间来储存块的信息，包括它的大小和状态（分配还是释放）。这些数据位叫做“边界标签”。使用这些标签， `malloc` 就可以从任何块移动到内存中上一个或下一个块。此外，空闲块会链接到一个双向链表中，所以每个空闲块也包含指向“空闲链表”中下一个块和上一个块的指针。边界标签和空闲链表指针构成了 `malloc` 的内部数据结构。这些数据结构穿插在程序的数据中，所以程序错误很容易破坏它们。
- 空间开销：边界标签和空闲链表指针也占据空间。最小的内存块大小在大多数系统上是16字节。所以对于非常小的内存块， `malloc` 在空间上并不高效。如果你的程序需要大量的小型数据结构，将它们分配在数组中可能更高效一些。
- 碎片：如果你以多种大小分配和释放块，堆区就会变得碎片化。也就是说，空闲空间会打碎成许多小型片段。碎片非常浪费空间，它也会通过使缓存效率低下来降低程序的速度。
- 装箱和缓存：空闲链表在箱子中以大小排序，所以当 `malloc` 搜索特定大小的内存块时，它知道应该在哪个箱子中寻找。所以如果你释放了一块内存，之后立即以相同大小分配

一块内存，`malloc`通常会很快。

第七章 缓存

> 作者：Allen B. Downey
>
> 原文：Chapter 7 Caching
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0

7.1 程序如何运行

为了理解缓存，你需要理解计算机如何运行程序。你应该学习计算机体系结构来深入理解这个话题。这一章中我的目标是给出一个程序执行的简单模型。

当程序启动时，代码（或者程序文本）通常位于硬盘上。操作系统创建新的进程来运行程序，之后“加载器”将代码从存储器复制到主存中，并且通过调用 `main` 来启动程序。

在程序运行之中，它的大部分数据都储存在主存中，但是一些数据在寄存器中，它们是CPU上的小型储存单元。这些寄存器包括：

- 程序计数器（PC），它含有程序下一条指令（在内存中）的地址。
- 指令寄存器（IR），它含有当前执行的指令的机器码。
- 栈指针（SP），它含有当前函数栈帧的指针，其中包含函数参数和局部变量。
- 程序当前使用的存放数据的通用寄存器。
- 状态寄存器，或者位寄存器，含有当前计算的信息。例如，位寄存器通常含有一位来存储上个操作是否是零的结果。

在程序运行之中，CPU执行下列步骤，叫做“指令周期”：

- 取指（Fetch）：从内存中获取下一条指令，储存在指令寄存器中。
- 译码（Decode）：CPU的一部分叫做“控制单元”，将指令译码，并向CPU的其它部分发送信号。
- 执行（Execute）：收到来自控制单元的信号后会执行合适的计算。

大多数计算机能够执行几百条不同的指令，叫做“指令集”。但是大多数指令可归为几个普遍的分类：

- 加载：将内存中的值送到寄存器。
- 算术/逻辑：从寄存器加载操作数，执行算术运算，并将结果储存到寄存器。
- 储存：将寄存器中的值送到内存。
- 跳转/分支：修改程序计数器，使控制流跳到程序的另一个位置。分支通常是有条件的，也就是说它会检查位寄存器中的旗标，只在设置时跳转。

一些指令集，包括普遍的x86，提供加载和算术运算的混合指令。

在每个指令周期中，指令从程序文本处读取。另外，普通程序中几乎一半的指令都用于储存或读取数据。计算机体系结构的一个基础问题，“内存瓶颈”就在这里。

在当前的台式机上，CPU通常为2GHz，也就是说每0.5ns就会初始化一条新的语句。但是它用于从内存中传送数据的时间约为10ns。如果CPU需要等10ns来抓取下一条指令，再等10ns来加载数据，它可能需要40个时钟周期来完成一条指令。

7.2 缓存性能

这一问题的解决方案，或者至少是一部分的解决方案，就是缓存。“缓存”是CPU上小型、快速的储存空间。在当前的计算机上，储存通常为1~2MiB，访问速度为1~2ns。

当CPU从内存中读取数据时，它将一份副本存到缓存中。如果再次读取相同的数据，CPU就直接读取缓存，不用再等待内存了。

当最后缓存满了的时候，为了能让新的数据进来，我们需要将一些数据扔掉。所以如果CPU加载数据之后，过了一段时间再来读取，数据就可能不在缓存中了。

许多程序的性能受限于缓存的效率。如果CPU所需的数据通常在缓存中，程序可以以CPU的全速来运行。如果CPU时常需要不在缓存中的数据，程序就会受限于内存的速度。

缓存的“命中率”`h`，是内存访问时，在缓存中找到数据的比例。“缺失率”`m`，是内存访问时需要访问内存的比例。如果`Th`是处理缓存命中的时间，`Tm`是缓存未命中的时间，每次内存访问的平均时间是：

```
h * Th + m * Tm
```

同样，我们可以定义“缺失惩罚”，它是处理缓存未命中所需的额外时间，`Tp = Tm - Th`，那么平均访问时间就是：

```
Th + m * Tp
```

当缺失率很低时平均访问时间趋近于`Th`，也就是说，程序可以表现为内存具有缓存的速度那样。

7.3 局部性

当程序首次读取某个字节时，缓存通常加载一“块”或一“行”数据，包含所需的字节和一些相邻数据。如果程序继续读取这些相邻数据，它们就已经在缓存中了。

例如，假设块大小是64B，你读取一个长度为64的字符串，字符串的首个字节恰好在块的开头。当你加载首个字节之后，你触发了缺失惩罚，但是之后字符串的剩余部分都在缓存中。在读取整个字符串之后，命中率是63/64。如果字符串被分在两个块中，你应该会触发两次缺失惩罚。但是这个命中率是62/64，约为97%。

另一方面，如果程序不可预测地跳来跳去，从内存中零散的位置读取数据，很少两次访问到相同的位置，缓存的性能就会很低。

程序使用相同数据多于一次的倾向叫做“时间局部性”。使用相邻位置的数据的倾向叫做“空间局部性”。幸运的是，许多程序天生就带有这两种局部性：

- 许多程序含有非跳转或分支的代码块。在这些代码块中指令顺序执行，访问模式具有空间局部性。
- 在循环中，程序执行多次相同指令，所以访问模式具有时间局部性。
- 一条指令的结果通常用于下一指令的操作数，所以数据访问模式具有时间局部性。
- 当程序执行某个函数时，它的参数和局部变量在栈上储存在一起。这些值的访问具有空间局部性。
- 最普遍的处理模型之一就是顺序读写数组元素。这一模式也具有空间局部性。

下一节中我们会探索程序的访问模式和缓存性能的关系。

7.4 缓存性能的度量

当我还是UC伯克利的毕业生时，我是Brian Harvey计算机体系结构课上的助教。我最喜欢的练习之一涉及到一个迭代数组，读写元素并度量平均时间的程序。通过改变数组的大小，就有可能推测出缓存的大小，块的大小，和一些其它属性。

我的这一程序的修改版本在本书仓库的 `cache` 目录下。

程序的核心部分是个循环：

```
iters = 0;
do {
    sec0 = get_seconds();

    for (index = 0; index < limit; index += stride)
        array[index] = array[index] + 1;

    iters = iters + 1;
    sec = sec + (get_seconds() - sec0);

} while (sec < 0.1);
```

内部的 `for` 循环遍历了数组。 `limit` 决定数组遍历的范围。 `stride` 决定跳过多少元素。例如，如果 `limit` 是16， `stride` 是4，循环就会访问0、4、8、和12。

`sec` 跟踪了CPU用于内循环的的全部时间。外部循环直到 `sec` 超过0.1秒才会停止，这对于我们计算出平均时间所需的精确度已经足够长了。

`get_seconds` 使用系统调用 `clock_gettime`，将结果换算成秒，并且以 `double` 返回结果。

```
double get_seconds(){
    struct timespec ts;
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &ts);
    return ts.tv_sec + ts.tv_nsec / 1e9;
}
```

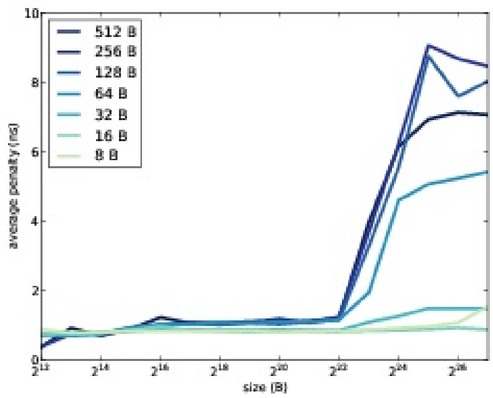


图 7.1：数据大小和步长的平均缺失惩罚函数

为了将访问数据的时间分离出来，程序运行了第二个循环，它除了内循环不访问数据之外完全相同。它总是增加相同的变量：

```
iters2 = 0;
do {
    sec0 = get_seconds();

    for (index = 0; index < limit; index += stride)
        temp = temp + index;

    iters2 = iters2 + 1;
    sec = sec - (get_seconds() - sec0);

} while (iters2 < iters);
```

第二个循环运行和第一个循环相同数量的迭代。在每轮迭代之后，它从 `sec` 中减少了消耗的时间。当循环完成时， `sec` 包含了所有数组访问的总时间，减去用于增加 `temp` 的时间。其中的差就是所有访问触发的全部缺失惩罚。最后，我们将它除以访问总数来获取每次访问的平均缺失惩罚，以ns为单位：

```
sec * 1e9 / iters / limit * stride
```

如果你编译并运行 `cache.c`，你应该看到这样的输出：

```
Size:     4096 Stride:       8 read+write: 0.8633 ns
Size:     4096 Stride:      16 read+write: 0.7023 ns
Size:     4096 Stride:      32 read+write: 0.7105 ns
Size:     4096 Stride:      64 read+write: 0.7058 ns
```

如果你安装了Python和Matplotlib，你可以使用 `graph_data.py` 来使结果变成图形。图7.1展示了我运行在Dell Optiplex 7010上的结果。要注意数组大小和步长以字节为单位表述，并不是数组元素数量。

花一分钟来考虑这张图片，并且看看你是否能推断出缓存信息。下面是一些需要思考的事情：

- 程序多次遍历并读取数组，所以有大量的时间局部性。如果整个数组能放进缓存，平均缺失惩罚应几乎为0。
- 当步长是4的时候，我们读取了数组的每个元素，所以程序有大量的空间局部性。例如，如果块大小足以包含64个元素，即使数组不能完全放在缓存中，命中率应为63/64。
- 如果步长等于块的大小（或更大），空间局部性应为0，因为每次我们读取一个块的时候，我们只访问一个元素。这种情况下，我们会看到最大的缺失惩罚。

总之，如果数组比缓存大小更小，或步长小于块的大小，我们认为会有良好的缓存性能。如果数组大于缓存大小，并且步长较大时，性能只会下降。

在图7.1中，只要数组小于 `2 ** 22` 字节，缓存性能对于所有步长都很好。我们可以推测缓存大小近似4MiB。实际上，根据规范应该是3MiB。

当步长为8、16或32B时，缓存性能良好。在64B时开始下降，对于更大的步长，平均缺失惩罚约为9ns。我们可以推断出块大小为128B。

许多处理器都使用了“多级缓存”，它包含一个小型快速的缓存，和一个大型慢速的缓存。这个例子中，当数组大小大于 `2 ** 14` B时，缺失惩罚似乎增长了一点。所以这个处理器可能也拥有一个访问时间小于1ns的16KB缓存。

7.5 缓存友好的编程

内存的缓存功能由硬件实现，所以多数情况下程序员都不需要知道太多关于它的东西。但是如果你知道缓存如何工作，你就可以编写更有效利用它们的程序。

例如，如果你在处理一个大型数组，只遍历数组一次，在每个元素上执行多个操作，可能比遍历数组多次要快。

如果你处理二维数组，它以行数组的形式储存。如果你需要遍历元素，按行遍历并且步长为元素大小会比按列遍历并且步长为行的大小更快。

链表数据结构并不总具有空间局部性，因为节点在内存中并不一定是连续的。但是如果你同时分配了很多个节点，它们在堆中通常分配到一起。或者，如果你一次分配了一个节点数组，你应该知道它们是连续的，这样会更好。

类似归并排序的递归策略通常具有良好的缓存行为，因为它们将大数组划分为小片段，之后处理这些小片段。有时这些算法可以调优来利用缓存行为。

对于那些性能至关重要的应用，可以设计适配缓存大小、块大小以及其它硬件特征的算法。像这样的算法叫做“缓存感知”。缓存感知算法的明显缺点就是它们硬件特定的。

7.6 存储器层次结构

在这一章的几个位置上，你可能会有一个问题：“如果缓存比主存快得多，那为什么不使用一大块缓存，然后把主存扔掉呢？”

在没有深入计算机体系结构之前，可以给出两个原因：电子和经济学上的。缓存很快是由于它们很小，并且离CPU很近，这可以减少由于电容造成的延迟和信号传播。如果你把缓存做得很大，它就变得很慢。

另外，缓存占据处理器芯片的空间，更大的处理器会更贵。主存通常使用动态随机访问内存（DRAM），每位上只有一个晶体管和一个电容，所以它可以将更多内存打包在同一空间上。但是这种实现内存的方法要比缓存实现的方式更慢。

同时主存通常包装在双列直插式内存模块（DIMM）中，它至少包含16个芯片。几个小型芯片比一个大型芯片更便宜。

速度、大小和成本之间的权衡是缓存的根本原因。如果有既快又大还便宜的内存技术，我们就不需要其它东西了。

与内存相同的原则也适用于存储器。闪存非常快，但是它们比硬盘更贵，所以它们就更小。磁带比硬盘更慢，但是它们可以储存更多东西，相对较便宜。

下面的表格展示了每种技术通常的访问时间、大小和成本。

设备	访问时间	通常大小	成本
寄存器	0.5 ns	256 B	?
缓存	1 ns	2 MiB	?
DRAM	10 ns	4 GiB	$10 / GiB
SSD	10 µs	100 GiB	$1 / GiB
HDD	5 ms	500 GiB	$0.25 / GiB
磁带	minutes	1–2 TiB	$0.02 / GiB

寄存器的数量和大小取决于体系结构的细节。当前的计算机拥有32个通用寄存器，每个都可以储存一个“字”。在32位计算机上，一个字为32位，4个字节。64位计算机上，一个字为64位，8个字节。所以寄存器文件的总容量是100~300字节。

寄存器和缓存的成本很难衡量。它们包含在芯片的成本中。但是顾客并不能直接了解到其成本。

对于表中的其它数据，我观察了计算机在线商店中，通常待售的计算机硬件规格。截至你读到这里为止，这些数据应该已经过时了，但是它们可以带给你在过去的某个时间上，一些关于性能和成本差距的概念。

这些技术构成了“存储器体系结构”。结构中每一级都比它上一级大而缓慢。某种意义上，每一级都作为其下一级的缓存。 你可以认为主存是持久化储存在SSD或HDD上的程序和数据的缓存。并且如果你需要处理磁带上非常大的数据集，你可以用硬盘缓存一部分数据。

7.7 缓存策略

存储器层次结构展示了一个考虑到缓存的框架。在结构的每一级中，我们都需要强调四个缓存的基本问题：

- 谁在层次结构中上移或下移数据？在结构的顶端，寄存器通常由编译器完成分配。CPU上的硬件管理内存的缓存。在执行程序或打开文件的过程中，用户可以将存储器上的文件隐式移动到内存中。但是操作系统也会将数据从内存移动回存储器。在层次结构的底端，管理员在磁带和磁盘之间显式移动数据。
- 移动了什么东西？通常，在结构顶端的块大小比底端要小。在内存的缓存中，通常块大小为128B。内存中的页面可能为4KiB，但是当操作系统从磁盘读取文件时，它可能会一次读10或100个块。
- 数据什么时候会移动？在多数的基本的缓存中，数据在首次使用时会移到缓存。但是许多缓存使用一些“预取”机制，也就是说数据会在显式请求之前加载。我们已经见过预取的一些形式了：在请求其一部分时加载整个块。
- 缓存中数据在什么地方？当缓存填满之后，我们不把一些东西扔掉就不可能放进一些东西。理想化来说，我们打算保留将要用到的数据，并替换掉不会用到的数据。

这些问题的答案构成了“缓存策略”。在靠近顶端的位置，缓存策略倾向于更简单，因为它们非常快，并由硬件实现。在靠近底端的位置，会有更多做决定的次数，并且设计良好的策略会有很大不同。

多数缓存策略基于历史重演的原则，如果我们有最近时期的信息，我们可以用它来预测不久的将来。例如，如果一块数据在最近使用了，我们认为它不久之后会再次使用。这个原则展示了一种叫做“最近最少使用”的策略，即LRU。它从缓存中移除最久未使用的数据块。更多话题请见缓存算法的维基百科。

7.8 页面调度

在带有虚拟内存的系统中，操作系统可以将页面在存储器和内存之间移动。像我在6.2中提到的那样，这种机制叫做“页面调度”，或者简单来说叫“换页”。

下面是工作流程：

1. 进程A调用`malloc`来分配页面。如果堆中没有所请求大小的空闲空间，`malloc`会调用`sbrk`向操作系统请求更多内存。
2. 如果物理内存中有空闲页，操作系统会将其加载到进程A的页表，创建新的虚拟内存有效范围。
3. 如果没有空闲页面，调度系统会选择一个属于进程B的“牺牲页面”。它将页面内容从内存复制到磁盘，之后修改进程B的页表来表示这个页面“被换出”了。
4. 一旦进程B的数据被写入，页面会重新分配给进程A。为了防止进程A读取进程B的数据，页面应被清空。
5. 此时`sbrk`的调用可以返回了，向`malloc`提供堆区额外的空间。之后`malloc`分配所请求的内存并返回。进程A可以继续执行。
6. 当进程A执行完毕，或中断后，调度器可能会让进程B继续执行。当它访问到被换出的页面时，内存管理器单元注意到这个页面是“无效”的，并且会触发中断。
7. 当操作系统处理中断时，它会看到页面被换出了，于是它将页面从磁盘传送到内存。
8. 一旦页面被换入之后，进程B可以继续执行。

当页面调度工作良好时，它可以极大提升物理内存的利用水平，允许更多进程在更少的空间内执行。下面是它的原因：

- 大多数进程不会用完所分配的内存。`text`段的许多部分都永远不会执行，或者执行一次就再也不用了。这些页面可以被换出而不会引发任何问题。
- 如果程序泄露了内存，它可能会丢掉所分配的空间，并且永远不会使用它了。通过将这些页面换出，操作系统可以有效填补泄露。
- 在多数系统中，有些进程像守护进程那样，多数时间下都是闲置的，只在特定场合被“唤醒”来响应事件。当它们闲置时，这些进程可以被换出。
- 另外，可能有许多进程运行同一个程序。这些进程可以共享相同的`text`段，避免在物理内存中保留多个副本。

如果你增加分配给所有进程的总内存，它可以超出物理内存的大小，并且系统仍旧运行良好。

在某种程度上是这样。

当进程访问被换出的页面时，就需要从磁盘获取数据，这会花费几个毫秒。这一延迟通常很明显。如果你将一个窗口闲置一段时间，之后切换回它，它可能会执行得比较慢，并且你可能在页面换入时会听到磁盘工作的声音。

像这样偶尔的延迟可能还可以接受，但是如果你拥有很多占据大量空间的进程，它们就会相互影响。当进程A运行时，它会收回进程B所需的页面，之后进程B运行时，它又会收回进程A所需的页面。当这种情况发生时，两个进程都会执行缓慢，系统会变得无法响应。这种我们不想看到的场景叫做“颠簸”。

理论上，操作系统应该通过检测调度和块上的增长来避免颠簸，或者杀掉进程直到系统能够再次响应。但是在我看来，多数系统都没有这样做，或者做得不好。它们通常让用户去限制物理内存的使用，或者尝试在颠簸发生时恢复。

第八章 多任务

> 作者：Allen B. Downey
>
> 原文：Chapter 8 Multitasking
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0

在当前的许多系统上，CPU包含多个核心，也就是说它可以同时运行多个进程。而且，每个核心都具有“多任务”的能力，也就是说它可以从一个进程快速切换到另一个进程，创造出同时运行许多进程的幻象。

操作系统中，实现多任务的这部分叫做“内核”。在坚果或者种子中，内核是最内层的部分，由外壳所包围。在操作系统各种，内核是软件的最底层，由一些其它层包围，包括称为“Shell”的界面。计算机科学家喜欢引喻。

究其本质，内核的工作就是处理中断。“中断”是一个事件，它会停止通常的指令周期，并且使执行流跳到称为“中断处理器”的特殊代码区域内。

当一个设备向CPU发送信号时，会发生硬件中断。例如，网络设备可能在数据包到达时会产生中断，或者磁盘驱动器会在数据传送完成时产生中断。多数系统也带有以固定周期产生中断的计时器。

软件中断由运行中的程序所产生。例如，如果一条指令由于某种原因没有完成，可能就会触发中断，便于这种情况可被操作系统处理。一些浮点数的错误，例如除零错误，会由中断处理。

当程序需要访问硬件设备时，会进行“系统调用”，它就像函数调用，除了并非跳到函数的起始位置，而是执行一条特殊的指令来触发中断，使执行流跳到内核中。内核读取系统调用的参数，执行所请求的操作，之后使被中断进程恢复运行。

8.1 硬件状态

中断的处理需要硬件和软件的配合。当中断发生时，CPU上可能正在运行多条指令，寄存器中也储存着数据。

通常硬件负责将CPU设为一致状态。例如，每条指令应该执行完毕，或者完全没有执行，不应该出现执行到一半的指令。而且，硬件也负责保存程序计数器（PC），便于内核了解从哪里恢复执行。

之后，中断处理器通常负责保存寄存器的上下文。为了完成工作，内核需要执行指令，这会修改数据寄存器和位寄存器。所以这个“硬件状态”需要在修改之前保存，并在被中断的进程恢复运行后复原。

下面是这一系列事件的大致过程：

1. 当中断发生时，硬件将程序计数器保存到一个特殊的寄存器中，并且跳到合适的中断处理器。
2. 中断处理器将程序计数器和位寄存器，以及任何打算使用的数据寄存器的内容储存到内存中。
3. 中断处理器运行处理中断所需的代码。
4. 之后它复原所保存寄存器的内容。最后，复原被中断进程的程序计数器，这会跳回到被中断的进程。

如果这一机制正常工作，被中断进程通常没有办法知道这是一个中断，除非它检测到了指令间的小变化。

8.2 上下文切换

中断处理器非常快，因为它们不需要保存整个硬件状态。它们只需要保存打算使用的寄存器。

但是当中断发生时，内核并不总会恢复被中断的进程。它可以选择切换到其它进程，这种机制叫做“上下文切换”。

通常，内核并不知道一个进程会用到哪个寄存器，所以需要全部保存。而且，当它切换到新的进程时，它可能需要清除储存在内存管理单元（MMU）中的数据。以及在上下文切换之后，它可能要花费一些时间，为新的进程将数据加载到缓存中。 出于这些因素，上下文切换相对较慢，大约是几千个周期或几毫秒。

在多任务的系统中，每个进程都允许运行一小段时间，叫做“时间片”或“quantum”。在上下文切换的过程中，内核会设置一些硬件计数器，它们会在时间片的末尾产生中断。当中断发生时，内核可以切换到另一个进程，或者允许被中断的进程继续执行。操作系统中做决策的这一部分叫做“调度器”。

8.3 进程的生命周期

当进程被创建时，操作系统会为进程分配包含进程信息的数据结构，称为“进程控制块”（PCB）。在其它方面，PCB跟踪进程的状态，这包括：

- 运行（Running），如果进程正在运行于某个核心上。
- 就绪（Ready），如果进程可以但没有运行，通常由于就绪进程数量大于内核的数量。
- 阻塞（Blocked），如果进程由于正在等待未来的事件，例如网络通信或磁盘读取，而不

能运行。
- 终止（Done）：如果进程运行完毕，但是带有没有读取的退出状态信息。

下面是一些可导致进程状态转换的事件：

- 一个进程在运行中的程序执行类似于 `fork` 的系统调用时诞生。在系统调用的末尾，新的进程通常就绪。之后调度器可能恢复原有的进程（“父进程”），或者启动新的进程（“子进程”）。
- 当一个进程由调度器启动或恢复时，它的状态从就绪变为运行。
- 当一个进程被中断，并且调度器没有选择使它恢复，它的状态从运行变成就绪。
- 如果一个进程执行不能立即完成的系统调用，例如磁盘请求，它会变为阻塞，并且调度器会选择另一个进程。
- 当类似于磁盘请求的操作完成时，会产生中断。中断处理器弄清楚哪个进程正在等待请求，并将它的状态从阻塞变为就绪。
- 当一个进程调用 `exit` 时，中断处理器在PCB中储存退出代码，并将进程的状态变为终止。

8.4 调度

就像我们在2.3节中看到的那样，一台计算机上可能运行着成百上千条进程，但是通常大多数进程都是阻塞的。大多数情况下，只有一小部分进程是就绪或者运行的。当中断发生时，调度器会决定那个进程应启动或恢复。

在工作站或笔记本上，调度器的首要目标就是最小化响应时间，也就是说，计算机应该快速响应用户的操作。响应时间在服务器上也很重要，但是调度器同时也可能尝试最大化吞吐量，它是单位时间内所完成的请求。

调度器通常不需要关于进程所做事情的大量信息，所以它基于一些启发来做决策：

- 进程可能被不同的资源限制。执行大量计算的进程是计算密集的，也就是说它的运行时间取决于得到了多少CPU时间。从网络或磁盘读取数据的进程是IO密集的，也就是说如果数据输入和输出更快的话，它就会更快，但是在更多CPU时间下它不会运行得更快。最后，与用户交互的程序，在大多数时间里可能都是阻塞的，用于等待用户的动作。操作系统有时可以将进程基于它们过去的行为分类，并做出相应的调度。例如，当一个交互型进程不再被阻塞，应该马上运行，因为用户可能正在等待回应。另一方面，已经运行了很长时间的CPU密集的进程可能就不是时间敏感的。
- 如果一个进程可能会运行较短的时间，之后发出了阻塞的请求，它可能应该立即运行，出于两个原因：（1）如果请求需要一些时间来完成，我们应该尽快启动它，（2）长时间运行的进程应该等待短时间的进程，而不是反过来。作为类比，假设你在做苹果馅饼。面包皮需要5分钟来准备，但是之后需要半个小时的冷却。而馅料需要20分钟来准备。如果你首先准备面包皮，你可以在其冷却时准备馅料，并且可以在35分钟之内做完。如果你先准备馅料，就会花费55分钟。

大多数调度器使用一些基于优先级的调度形式，其中每个进程都有可以调上或调下的优先级。当调度器运行时，它会选择最高优先级的就绪进程。

下面是决定进程优先级的一些因素：

- 具有较高优先级的进程通常运行较快。
- 如果一个进程在时间片结束之前发出请求并被阻塞，就可能是IO密集型程序或交互型程序，优先级应该升高。
- 如果一个进程在整个时间片中都运行，就可能是长时间运行的计算密集型程序，优先级应该降低。
- 如果一个任务长时间被阻塞，之后变为就绪，它应该提升为最高优先级，便于响应所等待的东西。
- 如果进程A在等待进程B的过程中被阻塞，例如，如果它们由管道连接，进程B的优先级应升高。
- 系统调用`nice`允许进程降低（但不能升高）自己的优先级，并允许程序员向调度器传递显式的信息。

对于运行普通工作负载的多数系统，调度算法对性能并没有显著的影响。简单的调度策略就足够好了。

8.5 实时调度

但是，对于与真实世界交互的程序，调度非常重要。例如，从传感器和控制马达读取数据的程序，可能需要以最小的频率完成重复的任务，并且以最大的响应时间对外界事件做出反应。这些需求通常表述为必须在“截止期限”之前完成的“任务”。

调度满足截止期限的任务叫做“实时调度”。对于一些应用，类似于Linux的通用操作系统可以被修改来处理实时调度。这些修改可能包括：

- 为控制任务的优先级提供更丰富的API。
- 修改调度器来确保最高优先级的进程在固定时间内运行。
- 重新组织中断处理器来保证最大完成时间。
- 修改锁和其它同步机制（下一章会讲到），允许高优先级的任务预先占用低优先级的任务。
- 选择保证最大完成时间的动态内存分配实现。

对于更苛刻的应用，尤其是实时响应是生死攸关的领域，“实时操作系统”提供了专用能力，通常比通用操作系统拥有更简单的设计。

第九章 线程

> 作者：Allen B. Downey
>
> 原文：Chapter 9 Threads
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0

当我在2.3节提到线程的时候，我说过线程就是一种进程。现在我会更仔细地解释它。

当你创建进程时，操作系统会创建一块新的地址空间，它包含 `text` 段、 `static` 段、和堆区。它也会创建新的“执行线程”，这包括程序计数器和其它硬件状态，以及运行时栈。

我们目前为止看到的进程都是“单线程”的，也就是说每个地址空间中只运行一个执行线程。在这一章中，你会了解“多线程”的进程，它在相同地址空间内拥有多个运行中的线程。

在单一进程中，所有线程都共享相同的 `text` 段，所以它们运行相同的代码。但是不同线程通常运行代码的不同部分。

而且，它们共享相同的 `static` 段，所以如果一个线程修改了某个全局变量，其它线程会看到改动。它们也共享堆区，所以线程可以共享动态分配的内存块。

但是每个线程都有它自己的栈。所以线程可以调用函数而不相互影响。通常，线程并不能访问其它线程的局部变量。

这一章的示例代码在本书的仓库中，在名为 `counter` 的目录中。有关代码下载的更多信息，请见第零章。

9.1 创建线程

C语言使用的所普遍的线程标准就是POSIX线程，简写为 `pthread` 。POSIX标准定义了线程模型和用于创建和控制线程的接口。多数UNIX的版本提供了POSIX的实现。

> 译者注：C11标准也提供了POSIX线程的实现。为了避免冲突，函数的前缀改为了 `thrd` 。

使用 `pthread` 就像使用大多数C标准库那样：

- 你需要将头文件包含到程序开头。
- 你需要编写调用 `pthread` 所定义函数的代码。
- 当你编译程序时，需要链接 `pthread` 库。

例如，我包含了下列头文件：

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
```

前两个是标准库，第三个就是`pthread`。为了在`gcc`中和`pthread`一起编译，你可以在命令行中使用`-l`选项：

```
gcc -g -O2 -o array array.c -lpthread
```

这会编译名为`array.c`的源文件，带有调试信息和优化，并链接`pthread`库，之后生成名为`array`的可执行文件。

9.2 创建线程

用于创建线程的`pthread`函数叫做`pthread_create`。下面的函数展示了如何使用它：

```
pthread_t make_thread(void *(*entry)(void *), Shared *shared)
{
  int n;
  pthread_t thread;

  n = pthread_create(&thread, NULL, entry, (void *)shared);
  if (n != 0) {
    perror("pthread_create failed");
    exit(-1);
  }
  return thread;
}
```

`make_thread`是一个包装，我编写它便于使`pthread_create`更加易用，并提供错误检查。

`pthread_create`的返回类型是`pthread_t`，你可以将其看做新线程的ID或者“句柄”。

如果`pthread_create`成功了，它会返回0，`make_pthread`也会返回新线程的句柄。如果出现了错误，`pthread_create`会返回错误代码，`make_thread`会打印错误消息并退出。

`pthread_create`的参数需要一些解释。从第二个开始，`Shared`是我定义的结构体，用于包含在两个线程之间共享的值。下面的`typedef`语句创建了这个新类型：

```
typedef struct {
  int counter;
} Shared;
```

这里，唯一的共享变量是`counter`，`make_shared`为`Shared`结构体分配空间，并且初始化其内容：

```
Shared *make_shared()
{
  int i;
  Shared *shared = check_malloc(sizeof (Shared));
  shared->counter = 0;
  return shared;
}
```

`entry` 的参数声明为 `void` 指针，但在这个程序中我们知道它是一个指向 `Shared` 结构体的指针，所以我们可以对其做相应转换，之后将它传给执行实际工作的 `child_code` 。

作为一个简单的示例， `child_code` 打印了共享计数器的值，并增加它。

```
void child_code(Shared *shared)
{
  printf("counter = %d\n", shared->counter);
  shared->counter++;
}
```

当 `child_code` 返回时， `entry` 调用了 `pthread_exit` ，它可以用于将一个值传递给回收（join）当前线程的线程。这里，子线程没有什么要返回的，所以我们传递了 `NULL` 。

最后，下面是创建子线程的代码：

```
int i;
pthread_t child[NUM_CHILDREN];

Shared *shared = make_shared(1000000);

for (i=0; i<NUM_CHILDREN; i++) {
    child[i] = make_thread(entry, shared);
}
```

`NUM_CHILDREN` 是用于定义子线程数量的编译期常量。 `child` 是线程句柄的数组。

9.3 回收线程

当一个线程希望等待其它线程执行完毕，它需要调用 `pthread_join` 。下面是我对 `pthread_join` 的包装：

```
void join_thread(pthread_t thread)
{
  int ret = pthread_join(thread, NULL);
  if (ret == -1) {
    perror("pthread_join failed");
    exit(-1);
  }
}
```

参数是你想要等待的线程句柄。这个包装所做的事情就是调用 `pthread_join` 之后检查结果。

任何线程都可以回收其它线程，但是多数普遍的情况下，父线程创建并回收所有子线程。我们继续使用上一节的例子，下面是等待子线程的代码：

```
for (i=0; i<NUM_CHILDREN; i++) {
    join_thread(child[i]);
}
```

这个循环一次等待一个子线程，以它们创建的顺序。没有办法来保证子线程按照顺序执行完毕，但是这个循环在它们不这样的时候也会正确执行。如果某个子线程迟于其它线程，这个循环会等待它，其它子线程也会在同时执行完毕。但是无论如何，所有子线程执行完毕后，循环才会退出。

如果你下载这本书的仓库，你可以在 `counter/counter.c` 中找到它。你可以像这样编译并运行它：

```
$ make counter
gcc -Wall counter.c -o counter -lpthread
$ ./counter
```

当我以5个子线程运行它时，我获得了如下输出：

```
counter = 0
counter = 0
counter = 1
counter = 0
counter = 3
```

当你运行它时，你可能得到了不同的结果。并且如果你再次运行它，你可能每次都得到不同的结果。到底发生了什么呢？

9.4 同步错误

上一个程序的问题就是，子线程访问了共享变量 `counter` ，不带任何同步机制，所以在任何线程增加 `counter` 之前，这些线程读取到了它的相同值。

下面是一个事件序列，这可以解释上一节的输出：

```
Child A reads 0
Child B reads 0
Child C reads 0
Child A prints   0
Child B prints   0
Child A sets counter=1
Child D reads 1
Child D prints   1
Child C prints   0
Child A sets counter=1
Child B sets counter=2
Child C sets counter=3
Child E reads 3
Child E prints   3
Child D sets counter=4
Child E sets counter=5
```

每次你运行这个程序的时候，线程都会在不同时间点上中断，或者调度器可能选择不同的线程来运行，所以时间序列和结果都是不同的。

假设我们需要强行规定一个顺序。例如，我们想让每个线程读到 `counter` 的不同值并增加它，让 `counter` 的值反映出执行 `child_code` 的线程数量。

为了达到这一要求，我们可以使用“互斥体”（mutex），它提供了互斥体对象，来保证一段代码是“互斥”的，也就是说，一次只有一个线程可以执行这段代码。

我编写了一个叫做 `mutex.c` 的小型模块，来提供互斥体对象。我会首先向你展示如何使用，之后再展示工作原理。

下面是 `child_code` 使用互斥体同步线程的版本：

```
void child_code(Shared *shared)
{
  mutex_lock(shared->mutex);
  printf("counter = %d\n", shared->counter);
  shared->counter++;
  mutex_unlock(shared->mutex);
}
```

在任何线程访问 `counter` 之前，它们需要“锁住”互斥体，这样可以阻塞住所有其它线程。假设线程A锁住互斥体，并且执行到 `child_code` 的中间位置。如果线程B到达并执行了 `mutex`，它会被阻塞。

当线程A执行完毕后，它执行了 `mutex_unlock`，它允许线程B继续执行。实际上，一次只有一个排队中的线程会执行 `child_code`，所以它们不会互相影响。当我以5个子线程运行这段代码时，我会得到：

```
counter = 0
counter = 1
counter = 2
counter = 3
counter = 4
```

这样就满足了要求。为了使这个方案能够工作，我向 `Shared` 结构体中添加了 `Mutex`：

```
typedef struct {
  int counter;
  Mutex *mutex;
} Shared;
```

之后在 `make_shared` 中初始化它：

```
Shared *make_shared(int end)
{
  Shared *shared = check_malloc(sizeof(Shared));
  shared->counter = 0;
  shared->mutex = make_mutex();    //-- this line is new
  return shared;
}
```

这一节的代码在 `counter_mutex.c` 中，`Mutex` 的定义在 `mutex.c` 中，我会在下一节解释它。

9.5 互斥体

我的 `Mutex` 的定义是 `pthread_mutex_t` 类型的包装，它定义在POSIX线程API中。

为了创建POSIX互斥体，你需要为 `pthread_mutex_t` 分配空间，之后调用 `pthread_mutex_init` 。

一个问题就是在这个API下， `pthread_mutex_t` 表现为结构体，所以如果你将它作为参数传递，它会复制，这会使互斥体表现不正常。你需要传递 `pthread_mutex_t` 的地址来避免这种情况。

我的代码更加容易正确使用。它定义了一个类型， `Mutex` ，它是 `pthread_mutex_t` 的更加可读的名称：

```
#include <pthread.h>

typedef pthread_mutex_t Mutex;
```

之后它定义了 `make_mutex` ，它为 `mutex` 分配空间并初始化：

```
Mutex *make_mutex()
{
  Mutex *mutex = check_malloc(sizeof(Mutex));
  int n = pthread_mutex_init(mutex, NULL);
  if (n != 0) perror_exit("make_lock failed");
  return mutex;
}
```

返回值是一个指针，你可以将其作为参数传递，而不会有非预期的复制。

对互斥体加锁和解锁的函数都是POSIX函数的简单包装：

```
void mutex_lock(Mutex *mutex)
{
  int n = pthread_mutex_lock(mutex);
  if (n != 0) perror_exit("lock failed");
}

void mutex_unlock(Mutex *mutex)
{
  int n = pthread_mutex_unlock(mutex);
  if (n != 0) perror_exit("unlock failed");
}
```

代码在`mutex.c`和头文件`mutex.h`中。

第十章 条件变量

> 作者：Allen B. Downey
>
> 原文：Chapter 10 Condition variables
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0

像上一章所展示的那样，许多简单的同步问题都可以用互斥体解决。这一章中我会介绍一个更大的挑战，著名的“生产者-消费者”问题，以及一个用于解决它的新工具，条件变量。

10.1 工作队列

在一些多线程的程序中，线程被组织用于执行不同的任务。通常它们使用队列来相互通信，其中一些线程叫做“生产者”，向队列中放入数据，另一些线程叫做“消费者”，从队列取出数据。

例如，在GUI应用中，可能有一个运行GUI的线程响应用户事件，而其它线程负责处理用户的请求。这里，GUI线程可能将数据放入队列中，而“后台”线程从队列中取出请求并执行。

为了支持这种组织，我们需要一个“线程安全”的队列实现，也就是说每个线程都可以同时访问队列。我们至少需要处理一个特殊情况，队列是空的，以及如果队列的大小有限制，队列是满的。

我会从一个非线程安全的简单队列开始，之后我们会观察其中的错误并修复它。这个示例的代码在本书仓库的`queue`目录中。`queue.c`文件包含了一个环形缓冲区的基本实现。你可以在环形缓冲区的维基百科查询更多信息。

下面是结构体的定义：

```
typedef struct {
  int *array;
  int length;
  int next_in;
  int next_out;
} Queue;
```

`array`是包含队列元素的数组。在这个例子中，元素都是整数，但是通常它们都是一些结构体，包含用户事件、工作项目以及其它。

`length`是数组的长度，`next_in`是数组的下标，用于索引下个元素应该添加到哪里；与之相似，`next_out`是应该被移除的下个元素的下标。

`make_queue` 为这个结构体分配空间，并且初始化所有字段：

```
Queue *make_queue(int length)
{
  Queue *queue = (Queue *) malloc(sizeof(Queue));
  queue->length = length;
  queue->array = (int *) malloc(length * sizeof(int));
  queue->next_in = 0;
  queue->next_out = 0;
  return queue;
}
```

`next_out` 的初始值需要一些解释。由于队列一开始为空，没有可移除的下一个元素，所以 `next_out` 是无效的。`next_out==next_in` 是个特殊情况，它表示队列为空，所以我们可以编写：

```
int queue_empty(Queue *queue)
{
  return (queue->next_in == queue->next_out);
}
```

现在我们可以使用 `queue_push` 向队列里面添加元素：

```
void queue_push(Queue *queue, int item) {
  if (queue_full(queue)) {
    perror_exit("queue is full");
  }

  queue->array[queue->next_in] = item;
  queue->next_in = queue_incr(queue, queue->next_in);
}
```

如果队列满了，`queue_push` 打印出错误信息并退出，我之后会解释 `queue_full` 。

如果队列没有满，`queue_push` 插入新元素，之后使用 `queue_incr` 增加 `next_in` ：

```
int queue_incr(Queue *queue, int i)
{
  return (i+1) % queue->length;
}
```

当索引 `i` 到达队列末尾时，它会转换为0。于是这样就很微妙了。如果我们持续向队列添加元素，最后 `next_in` 会赶上 `next_out` 。但是如果 `next_in == next_out` 我们会错误地认为队列是空的。

为了避免这种情况，我们定义另一种特殊情况来表示队列是满的：

```
int queue_full(Queue *queue)
{
  return (queue_incr(queue, queue->next_in) == queue->next_out);
}
```

如果 `next_in` 增加后与 `next_out` 重合，那么我们如果添加新的元素，就会使队列看起来是空的。所以我们在“末尾”留出一个元素（要记住队列的末尾可能位于任何地方，不一定是数组末尾）。

现在我们可以编写 `queue_pop` ，它移除并返回队列的下一个元素：

```
int queue_pop(Queue *queue) {
  if (queue_empty(queue)) {
    perror_exit("queue is empty");
  }

  int item = queue->array[queue->next_out];
  queue->next_out = queue_incr(queue, queue->next_out);
  return item;
}
```

如果你尝试从空队列中弹出元素， `queue_pop` 会打印错误信息并退出。

10.2 生产者和消费者

现在让我们创建一些访问这个队列的线程。下面是生产者的代码：

```
void *producer_entry(void *arg)
{
  int i;
  Shared *shared = (Shared *) arg;

  for (i=0; i<QUEUE_LENGTH-1; i++) {
    printf("adding item %d\n", i);
    queue_push(shared->queue, i);
  }
  pthread_exit(NULL);
}
```

下面是消费者的代码：

```
void *consumer_entry(void *arg)
{
  int i;
  int item;
  Shared *shared = (Shared *) arg;

  for (i=0; i<QUEUE_LENGTH-1; i++) {
    item = queue_pop(shared->queue);
    printf("consuming item %d\n", item);
  }
  pthread_exit(NULL);
}
```

下面是用于启动线程并等待它们的主线程代码：

```
int i;
pthread_t child[NUM_CHILDREN];

Shared *shared = make_shared();

child[0] = make_thread(producer_entry, shared);
child[1] = make_thread(consumer_entry, shared);

for (i=0; i<NUM_CHILDREN; i++) {
    join_thread(child[i]);
}
```

最后，下面是包含队列的共享结构：

```
typedef struct {
  Queue *queue;
} Shared;

Shared *make_shared()
{
  Shared *shared = check_malloc(sizeof(Shared));
  shared->queue = make_queue(QUEUE_LENGTH);
  return shared;
}
```

到目前为止我们所写的代码是一个好的开始，但是有如下几种问题：

- 队列的访问不是线程安全的。不同的线程能同时访问 `array` 、 `next_in` 和 `next_out` ，并且会使队列处于损坏的、“不一致”的状态。
- 如果消费者首先被调度，它会发现队列为空，打印错误信息并退出。我们应该阻塞住消费者，直到队列非空。与之相似，我们应该在队列满了的情况下阻塞住生产者。

在下一节中，我们会使用互斥体解决这一个问题。之后的章节中我们会使用条件变量解决第二个问题。

10.3 互斥体

我们可以使用互斥体使队列线程安全。这个版本的代码在 `queue_mutex.c` 中。

首先我们向队列结构中添加一个互斥体指针：

```
typedef struct {
  int *array;
  int length;
  int next_in;
  int next_out;
  Mutex *mutex;          //-- this line is new
} Queue;
```

之后在 `make_queue` 中初始化互斥体：

```
Queue *make_queue(int length)
{
  Queue *queue = (Queue *) malloc(sizeof(Queue));
  queue->length = length;
  queue->array = (int *) malloc(length * sizeof(int));
  queue->next_in = 0;
  queue->next_out = 0;
  queue->mutex = make_mutex();   //-- new
  return queue;
}
```

接下来向 `queue_push` 添加同步代码：

```
void queue_push(Queue *queue, int item) {
  mutex_lock(queue->mutex);   //-- new
  if (queue_full(queue)) {
    mutex_unlock(queue->mutex);   //-- new
    perror_exit("queue is full");
  }

  queue->array[queue->next_in] = item;
  queue->next_in = queue_incr(queue, queue->next_in);
  mutex_unlock(queue->mutex);   //-- new
}
```

在检查队列是否已满之前，我们需要锁住互斥体。如果队列是满的，我们需要在退出之前解锁互斥体。否则线程应该保持互斥体锁住，使其它线程不能前进。

`queue_pop` 的同步代码与之相似：

```
int queue_pop(Queue *queue) {
  mutex_lock(queue->mutex);
  if (queue_empty(queue)) {
    mutex_unlock(queue->mutex);
    perror_exit("queue is empty");
  }

  int item = queue->array[queue->next_out];
  queue->next_out = queue_incr(queue, queue->next_out);
  mutex_unlock(queue->mutex);
  return item;
}
```

要注意其它队列函数，`queue_full`、`queue_empty` 和 `queue_incr` 都不需要锁住互斥体。任何调用这些函数的线程都需要首先锁住互斥体。这些要求是这些函数的接口文档的一部分。

使用这些额外的代码，队列就线程安全了。如果你运行它，你不会看到任何的同步错误。但是似乎消费者会在某个时间上退出，因为队列是空的。或者生产者会由于队列是满足而退出。

下一步就是添加条件变量。

10.4 条件变量

条件变量是条件相关的数据结构。它允许线程在某些条件变为真之前被阻塞。例如，`thread_push` 可能希望检查队列是否已满，如果是这样，就在队列未满之前阻塞。所以我们感兴趣的“条件”就是“队列未满”。

与之相似，`thread_pop` 希望等待“队列非空”的条件。

下面是我们向代码添加这些功能的方式。首先我们向队列结构中添加两个条件变量：

```
typedef struct {
  int *array;
  int length;
  int next_in;
  int next_out;
  Mutex *mutex;
  Cond *nonempty;   //-- new
  Cond *nonfull;    //-- new
} Queue;
```

之后在 `make_queue` 中初始化它们：

```
Queue *make_queue(int length)
{
  Queue *queue = (Queue *) malloc(sizeof(Queue));
  queue->length = length;
  queue->array = (int *) malloc(length * sizeof(int));
  queue->next_in = 0;
  queue->next_out = 0;
  queue->mutex = make_mutex();
  queue->nonempty = make_cond();   //-- new
  queue->nonfull = make_cond();    //-- new
  return queue;
}
```

现在在 `queue_pop` 中，如果我们发现队列为空，我们不要退出，而是使用条件变量来阻塞：

```
int queue_pop(Queue *queue) {
  mutex_lock(queue->mutex);
  while (queue_empty(queue)) {
    cond_wait(queue->nonempty, queue->mutex);  //-- new
  }

  int item = queue->array[queue->next_out];
  queue->next_out = queue_incr(queue, queue->next_out);
  mutex_unlock(queue->mutex);
  cond_signal(queue->nonfull);   //-- new
  return item;
}
```

`cond_wait` 有点复杂，所以让我们慢慢来。第一个参数是条件变量。这里我们需要等待的条件是“队列非空”。第二个变量是保护队列的互斥体。在你调用 `cond_wait` 之前，你需要先锁住互斥体，否则它不会生效。

当锁住互斥体的线程调用 `cond_wait` 时，它首先解锁互斥体，之后阻塞。这非常重要。如果 `cond_wait` 不在阻塞之前解锁互斥体，其它线程就不能访问队列，不能添加任何物品，队列会永远为空。

所以当消费者阻塞在 `nonempty` 的时候，生产者也可以运行。让我们来观察生产者运行 `queue_push` 时会发生什么：

```
void queue_push(Queue *queue, int item) {
  mutex_lock(queue->mutex);
  while (queue_full(queue)) {
    cond_wait(queue->nonfull, queue->mutex);    //-- new
  }

  queue->array[queue->next_in] = item;
  queue->next_in = queue_incr(queue, queue->next_in);
  mutex_unlock(queue->mutex);
  cond_signal(queue->nonempty);  //-- new
}
```

让我们假设队列现在未满，于是生产者并不会调用 `cond_wait` 也不会阻塞。它会向队列添加新的元素并解锁互斥体。但是在退出之前，它做了额外的一件事：它向 `nonempty` 条件变量发送信号。

向条件变量发送更新好表示条件为真，或者至少它可能为真。如果没有任何线程在等待条件变量，信号就不起作用。

如果有线程在等待条件变量，它们全部会从 `cond_wait` 解除阻塞并且恢复执行。但是在被唤醒的进程从 `cond_wait` 返回之前，它需要等待并再次锁住互斥体。

现在我们回到 `queue_pop` 来观察当线程从 `cond_wait` 返回时会发生什么。它会循环到 `while` 语句的开头，并再次检查条件。我会在之后解释其原因，但是现在让我们假设条件为真，也就是说队列非空。

当线程从 `while` 循环退出之后，我们知道了两件事情：（1）条件为真，所以队列中至少有一个物品，（2）互斥体是锁住的，所以访问队列是安全的。

在移除物品之后，`queue_pop` 解锁了互斥体，发送了队列未满的信号，之后退出。

在下一节我会向你展示我的 `Cond` 的工作原因，但是首先我想回答两个常见问题：

- 为什么 `cond_wait` 在 `while` 循环中，而不是 `if` 语句中？也就是说，为什么在从 `cond_wait` 返回之后要再次检查条件？

  需要再次检查条件的首要原因就是信号拦截的可能性。假设线程A在等待 `nonempty` ，线程B向队列添加元素，之后向 `nonempty` 发送信号。线程A被唤醒并且尝试锁住互斥体，但是在轮到它之前，邪恶的线程C插进来了，锁住了互斥体，从队列中弹出物品并且解锁了互斥体。现在队列再次为空，但是线程A没有被阻塞。线程A会锁住互斥体并且从 `cond_wait` 返回。如果线程A不再次检查条件，它会尝试从空队列中弹出元素，可能会产生错误。

  > 译者注：有些条件变量的实现可以每次只唤醒一个线程，比如Java对象的 `notify` 方法。这种情况就可以使用 `if` 。

- 当人们了解条件变量时，另一个问题是“条件变量怎么知道它关联了哪个条件？”

  这一问题可以理解，因为在 `Cond` 结构和有关条件之间没有明显的关联。在它的使用方式中，关联是隐性的。

  下面是一种理解它的办法：当你调用 `cond_wait` 时，`Cond` 所关联的条件为假；当你调用 `cond_signal` 时它为真。当然，可能有一些条件第一种情况下为真，第二种情况下为假。正确的情况只在程序员的脑子中，所以它应该在文档中有详细的解释。

10.5 条件变量的实现

我在上一节中使用的条件变量是 `pthread_cond_t` 类型的包装，它定义在POSIX线程API中。这非常类似于 `Mutex`，它是 `pthread_mutex_t` 的包装。两个包装都定义在 `utils.c` 和 `utils.h` 中。

下面是类型定义：

```
typedef pthread_cond_t Cond;
```

`make_cond` 分配空间，初始化条件变量，之后返回指针：

```
Cond *make_cond()
{
  Cond *cond = check_malloc(sizeof(Cond));
  int n = pthread_cond_init(cond, NULL);
  if (n != 0) perror_exit("make_cond failed");

  return cond;
}
```

下面是 `cond_wait` 和 `cond_signal` 的包装：

```
void cond_wait(Cond *cond, Mutex *mutex)
{
  int n = pthread_cond_wait(cond, mutex);
  if (n != 0) perror_exit("cond_wait failed");
}

void cond_signal(Cond *cond)
{
  int n = pthread_cond_signal(cond);
  if (n != 0) perror_exit("cond_signal failed");
}
```

到这里就应该没有什么意外的东西了。

第十一章 C语言中的信号量

> 作者：Allen B. Downey
>
> 原文：Chapter 11 Semaphores in C
>
> 译者：飞龙
>
>

信号量是学习同步的一个好方式，但是它们实际上并没有像互斥体和条件变量一样被广泛使用。

尽管如此，还是有一些同步问题可以用信号量简单解决，产生显然更加合适的解决方案。

这一章展示了C语言用于处理信号量的API，以及我用于使它更加容易使用的代码。而且它展示了一个终极挑战：你能不能使用互斥体和条件变量来实现一个信号量？

这一章的代码在本书仓库的`semaphore`目录中。

11.1 POSIX信号量

信号量是用于使线程协同工作而不互相影响的数据结构。

POSIX标准规定了信号量的接口，它并不是`pthread`的一部分，但是多数实现`pthread`的UNIX系统也实现了信号量。

POSIX信号量的类型是`sem_t`。这个类型表现为结构体，所以如果你将它赋值给一个变量，你会得到它的内容副本。复制信号量完全是一个坏行为，在POSIX中，它的复制行为是未定义的。

幸运的是，包装`sem_t`使之更安全并易于使用相当容易。我的包装API在`sem.h`中：

```c
typedef sem_t Semaphore;

Semaphore *make_semaphore(int value);
void semaphore_wait(Semaphore *sem);
void semaphore_signal(Semaphore *sem);
```

`Semaphore`是`sem_t`的同义词，但是我认为它更加可读，而且大写的首字母会提醒我将它当做对象并使用指针传递它。

这些函数的实现在`sem.c`中：

```
Semaphore *make_semaphore(int value)
{
  Semaphore *sem = check_malloc(sizeof(Semaphore));
  int n = sem_init(sem, 0, value);
  if (n != 0) perror_exit("sem_init failed");
  return sem;
}
```

`make_semaphore`接收信号量的初始值作为参数。它为信号量分配空间，将信号量初始化，之后返回指向`Semaphore`的指针。

如果执行成功，`sem_init`返回0；如果有任何错误，它返回-1。使用包装函数的一个好处就是你可以封装错误检查代码，这会使使用这些函数的代码更加易读。

下面是`semaphore_wait`的实现：

```
void semaphore_wait(Semaphore *sem)
{
  int n = sem_wait(sem);
  if (n != 0) perror_exit("sem_wait failed");
}
```

下面是`semaphore_signal`：

```
void semaphore_signal(Semaphore *sem)
{
  int n = sem_post(sem);
  if (n != 0) perror_exit("sem_post failed");
}
```

我更喜欢把这个这个操作叫做“signal”而不是“post”，虽然它们是一个意思（发射）。

> 译者注：如果你习惯了互斥体（锁）的操作，也可以改成`lock`和`unlock`。互斥体其实就是信号量容量为1时的特殊形态。

下面是一个例子，展示了如何将信号量用作互斥体：

```
Semaphore *mutex = make_semaphore(1);
semaphore_wait(mutex);
// protected code goes here
semaphore_signal(mutex);
```

当你将信号量用作互斥体时，通常需要将它初始化为1，来表示互斥体是未锁的。也就是说，只有一个线程可以通过信号量而不被阻塞。

这里我使用了变量名称`mutex`来表明信号量被用作互斥体。但是要记住信号量的行为和`pthread`互斥体不完全相同。

11.2 使用信号量解决生产者-消费者问题

使用这些信号量的包装函数，我们可以编写出生产者-消费者问题的解决方案。这一节的代码在`queue_sem.c`。

下面是`Queue`的一个新定义，使用信号量来代替互斥体和条件变量：

```
typedef struct {
  int *array;
  int length;
  int next_in;
  int next_out;
  Semaphore *mutex;       //-- new
  Semaphore *items;       //-- new
  Semaphore *spaces;      //-- new
} Queue;
```

下面是`make_queue`的新版本：

```
Queue *make_queue(int length)
{
  Queue *queue = (Queue *) malloc(sizeof(Queue));
  queue->length = length;
  queue->array = (int *) malloc(length * sizeof(int));
  queue->next_in = 0;
  queue->next_out = 0;
  queue->mutex = make_semaphore(1);
  queue->items = make_semaphore(0);
  queue->spaces = make_semaphore(length-1);
  return queue;
}
```

`mutex`用于确保队列的互斥访问，初始值为1，说明互斥体最开始是未锁的。

`item`是队列中物品的数量，它也是可非阻塞执行`queue_pop`的消费者线程的数量。最开始队列中没有任何物品。

`spaces`是队列中剩余空间的数量，也是可非阻塞执行`queue_push`的线程数量。最开始的空间数量就是队列的容量`length - 1`。

下面是`queue_push`的新版本，它由生产者线程调用：

```
void queue_push(Queue *queue, int item) {
  semaphore_wait(queue->spaces);
  semaphore_wait(queue->mutex);

  queue->array[queue->next_in] = item;
  queue->next_in = queue_incr(queue, queue->next_in);

  semaphore_signal(queue->mutex);
  semaphore_signal(queue->items);
}
```

要注意`queue_push`并不需要调用`queue_full`，因为信号量跟踪了有多少空间可用，并且在队列满了的时候阻塞住生产者。

下面是`queue_pop`的新版本：

```
int queue_pop(Queue *queue) {
  semaphore_wait(queue->items);
  semaphore_wait(queue->mutex);

  int item = queue->array[queue->next_out];
  queue->next_out = queue_incr(queue, queue->next_out);

  semaphore_signal(queue->mutex);
  semaphore_signal(queue->spaces);

  return item;
}
```

这个解决方案在《The Little Book of Semaphores》中的第四章以伪代码解释。

为了使用本书仓库的代码，你需要编译并运行这个解决方案，你应该执行：

```
$ make queue_sem
$ ./queue_sem
```

11.3 编写你自己的信号量

任何可以使用信号量解决的问题也可以使用条件变量和互斥体来解决。一个证明方法就是可以使用条件变量和互斥体来实现信号量。

在你继续之前，你可能想要将其做为一个练习：编写函数，使用条件变量和互斥体实现 `sem.h` 中的信号量API。你可以将你的解决方案放到本书仓库的 `mysem.c` 和 `mysem.h` 中，你会在 `mysem_soln.c` 和 `mysem_soln.h` 中找到我的解决方案。

如果你在开始时遇到了麻烦，你可以使用下面来源于我的代码的结构体定义，作为提示：

```
typedef struct {
  int value, wakeups;
  Mutex *mutex;
  Cond *cond;
} Semaphore;
```

`value` 是信号量的值。`wakeups` 记录了挂起信号的数量，也就是说它是已被唤醒但是还没有恢复执行的线程数量。`wakeups` 的原因是确保我们的信号量拥有《The Little Book of Semaphores》中描述的性质3。

`mutex` 提供了 `value` 和 `wakeups` 的互斥访问，`cond` 是线程在需要等待信号量时所等待的条件变量。

下面是这个结构体的初始化代码：

```
Semaphore *make_semaphore(int value)
{
  Semaphore *semaphore = check_malloc(sizeof(Semaphore));
  semaphore->value = value;
  semaphore->wakeups = 0;
  semaphore->mutex = make_mutex();
  semaphore->cond = make_cond();
  return semaphore;
}
```

11.3.1 信号量的实现

下面是我使用POSIX互斥体和条件变量的信号量实现：

```
void semaphore_wait(Semaphore *semaphore)
{
  mutex_lock(semaphore->mutex);
  semaphore->value--;

  if (semaphore->value < 0) {
    do {
      cond_wait(semaphore->cond, semaphore->mutex);
    } while (semaphore->wakeups < 1);
    semaphore->wakeups--;
  }
  mutex_unlock(semaphore->mutex);
}
```

当线程等待信号量时，需要在减少 `value` 之前锁住互斥体。如果信号量的值为负，线程会被阻塞直到 `wakeups` 可用。要注意当它被阻塞时，互斥体是未锁的，所以其它线程可以向条件变量发送信号。

`semaphore_signal` 的代码如下：

```
void semaphore_signal(Semaphore *semaphore)
{
  mutex_lock(semaphore->mutex);
  semaphore->value++;

  if (semaphore->value <= 0) {
    semaphore->wakeups++;
    cond_signal(semaphore->cond);
  }
  mutex_unlock(semaphore->mutex);
}
```

同样，线程在增加 `value` 之前需要锁住互斥体。如果信号量是负的，说明还有等待线程，所以发送线程需要增加 `wakeups` 并向条件变量发送信号。

此时等待线程可能会唤醒，但是互斥体仍然会锁住它们，直到发送线程解锁了它。

这个时候，某个等待线程从 `cond_wait` 中返回，之后检查是否 `wakeup` 仍然有效。如果没有它会循环并再次等待条件变量。如果有效，它会减少 `wakeup` ，解锁互斥体并退出。

这个解决方案使用`do-while`循环的原因可能并不是很明显。你知道为什么不使用更普遍的`while`循环吗？会出现什么问题呢？

问题就是`while`循环的实现不满足性质3。一个发送线程可以在之后的运行中收到它自己的信号。

使用`do-while`循环，就确保[1]了当一个线程发送信号时，另一个等待线程会收到信号，即使发送线程在某个等待线程恢复之前继续运行并锁住互斥体。

> [1] 好吧，几乎是这样。实际上一个时机恰当的虚假唤醒会打破这一保证。

捐赠名单

感谢以下童鞋的捐助，你们的慷慨是我继续的动力：

donor	value
Michael翔	1.88
justjavac	50.00